FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Serious Game for Learning About Software Architecture and Design

João Miguel Dias Ferreira Gouveia

DISSERTATION

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Nuno Honório Rodrigues Flores

July 28th, 2017

Serious Game for Learning About Software Architecture and Design

João Miguel Dias Ferreira Gouveia

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Dr. Hugo José Sereno Lopes Ferreira

External Examiner: Dr. Ângelo Martins

Supervisor: Dr. Nuno Honório Rodrigues Flores

July 28th, 2017

Resumo

Arquitetura e Design de Software (***SAD***) pode ser percebido como um subtópico do domínio de sabedoria conhecido como "Engenharia de Software" (***SE***) e também como uma área com uma variedade ampla de conceitos e sabedoria com várias aplicações. Jogos sérios são feitos com a intenção clara de chamar assuntos sérios à atenção sem perder o factor de "diversão" e são frequentemente aplicados a experiências educacionais. Existem jogos sérios para algumas áreas de SE, no entanto, não foram encontrados jogos digitais educacionais até agora. A filosofia de design de um jogo eficaz para um tópico SE é baseada em funções de aprendizagem e ensino (***LTFs***) que podem ser convertidas em padrões de design de jogos (***GDPs***). Um subconjunto dessas mesmas LTFs são dedicadas a tópicos SE que, por sua vez, conduzem a certos GDPs. Com isto em mente, é possível provar que os GDPs previamente mencionados funcionam para o tópico de Arquitetura e Design de Software ao usá-los para desenvolver e implementar um jogo sério à volta desse tópico com êxito. O jogo em questão é um jogo com 5 níveis chamado Codebase Escape, onde o objetivo é completar os níveis ao desbloquear e depois responder a um quiz relacionado com SAD no fim de cada nível com êxito. O jogo foi validado através de um estudo empírico com estudantes com, supostamente, nenhum conhecimento do tópico de SAD. Os estudantes em questão foram divididos em dois grupos de tamanho semelhante, onde o primeiro grupo jogou o jogo e o segundo grupo não jogou o jogo. Depois, os dois grupos responderam um questionário sobre o tópico, onde a diferença de conhecimento entre os grupos foi medida. No contexto da experiência, o desfecho ideal é o primeiro grupo obter um desempenho melhor no questionário que o segundo grupo. O resultado do projeto foi que um jogo educativo para SAD aceitável mas com alguns problemas foi desenvolvido e o subconjunto existente de GDPs para educação de SE foi algo validado.

Palavras-chave

Jogo sério; software; arquitetura de software; design de software; engenharia de software.

Categorias

Software e a sua engenharia - Organização e propriedades de software - Domínios de software contextuais - Software de mundos digitais - Jogos interativos

Software e a sua engenharia - Organização e propriedades de software - Estruturas de sistemas de software - Arquiteturas de software

Software e a sua engenharia - Criação e gestão de software - Design de software

Abstract

Software Architecture and Design (***SAD***) can be understood as a subtopic of the "Software Engineering" (***SE***) domain of knowledge and also as a domain area with a wide range of concepts and knowledge with various applications.

Serious games are made with the clear intention of addressing serious issues without losing the "fun" factor and are often applied to educational experiences. There are serious games for some SE fields, however, no educational digital games have been found for SAD thus far.

The design philosophy of an effective game for a SE topic is based on learning and teaching functions (***LTFs***) that are able to be converted to game design patterns (***GDPs***). A subset of these same LTFs are dedicated to SE topics which, in turn, translate to certain GDPs. With this in mind, it is possible to prove that the aforementioned GDPs work for the topic of Software Architecture and Design by using them to successfully design and implement a serious game revolving around that topic.

The game in question is a game with 5 levels named *Codebase Escape*, where the goal is to clear the levels by successfully unlocking and then answering a quiz related to SAD at the end of each level. The game was validated through an empirical study with students with, supposedly, no knowledge on the topic of SAD. The students in question were divided into two groups of similar size, where the first group played the game and the second group did not. The two groups then answered a questionnaire about the topic, where the knowledge gap between the groups was measured. In the context of the experiment, the preferred outcome is that the first group performs better at the questionnaire than the second group.

The project's results were that a serviceable but flawed educational game for SAD was designed and the existing subset of GDPs for SE education was somewhat validated.

Keywords

Serious game; software; software architecture; software design; software engineering.

Categories

Software and its engineering - Software organization and properties - Contextual software domains - Virtual worlds software - Interactive games

Software and its engineering - Software organization and properties - Software system structures - Software architectures

Software and its engineering - Software creation and management - Designing software

Acknowledgements

I wish to express my sincere gratitude towards my family, my friends, the FEUP institution and everyone else who supported me throughout the long and arduous journey to get to where I am right now.

João Miguel Dias Ferreira Gouveia

Contents

List of Figures

List of Tables

Abbreviations

SE	Software Engineering
SAD	Software Architecture and Design
ESWS	Empirical Study With Students
ADL	Architecture Description Language
C&C	Component-and-Connector
LTF	Learning and Teaching Function
GDP	Game Design Pattern
SG	Serious Game
LOC	Line of Code
DOC	Documentation File
BBOM	*Big Ball of Mud*
HP	Health Point
BG	Background Quiz
KW	Knowledge Quiz
EF	External Factors Quiz
OS	Overall Satisfaction Quiz

Chapter 1

Introduction

Software is a key component of modern technology. It is built into hardware such as computers, smartphones and other machines and, as such, it is responsible for the appropriate performance of said hardware. Naturally, it is a worthy affair to know about how software is organized in the current age, and the field that specifies in this matter is the field of ***Software Architecture and Design*** (***SAD***).

Software Architecture and Design can be understood as a subtopic of the Software Engineering (***SE***) domain of knowledge. In itself, it is a domain area with a wide range of concepts and knowledge with various possible applications, as well as a cornerstone for the life cycle of software development and, as such, essential for developers [SKA15].

A way to make the process of introducing people to this topic in a way that potentially makes the process interesting and engaging is through the subgenre of games known an ***serious games*** (***SGs***). Serious games are games made with the clear intention of addressing serious issues that can specifically be utilized for education, in order to overcome specific learning issues.

The main objective of this project was to develop an effective serious game that teaches players about the topic of software architecture and design and to later validate it through an empirical study with students (***ESWS***).

The expected results of the project were that an effective educational game for SAD would be designed and the existing subset of GDPs for SE education were properly validated.

1.1 How to Read This Document

The document's structure from this point forward is as follows:

- Chapter 2, called State of the Art, details all the information that was used for the project and in the project's context;

- Chapter 3, named Serious Game for Learning About Software Architecture and Design, describes the open issues that are relevant to the topic at hand, the specific problem the project aimed to solve, the project itself and how the project attempted to solve the problem;
- Chapter 4, named Empirical Study With Students, explains the methodology used to validate the serious game, with testing of external factors included;
- Chapter 5, called Conclusion, outlines the overall contributions of this project to the issues it attempted to solve, as well as future work to be done on this topic.

Chapter 2

State of the Art

As the years passed, software engineering became an area of interest to a lot of people, and efforts to teach people about that area increased. There are multiple subgenres within software engineering, such as software design, software testing and, most importantly, software architecture and design, which is the topic the serious game that constitutes the bulk of this project will be about.

In order to demonstrate the level of knowledge that went into the game itself, this thesis will cover the field of software architecture and design, as well as serious games and how to effectively develop one.

2.1 Software Architecture

First of all, we must address the most prominent field the game is designed around: software architecture.

2.1.1 Definition of "Software Architecture"

Software architecture has various possible definitions. Some of these definitions are:

- A set of architectural elements of a software system that have a particular form [PW92];
- The specification of the structure of a software system [GS93];
- A software system's gross organization as a collection of interacting components [Gar00].

Software architecture is specifically concerned with higher-level, structural issues [MT10]. These issues include, but are not limited to [GS93]:

- Gross organization and global control structure;
- Protocols for communication;
- Synchronization and data access;

- Assignment of functionality to design elements;
- Composition of design elements;
- Scaling and performance;
- Physical distribution;
- Selection among design alternatives.

As software systems proceeded to get bigger and more complex over time, it became necessary to solve these issues. As such, it is by implementing a robust structure using software architecture that we make sure that these issues are properly dealt with and the software systems in question are adequately built.

However, the mere act of recognizing the necessity of software systems to maintain a well-groomed structure is insufficient. Rather, it is also of paramount importance to recognize the impact that said well-groomed structure carries on the system, both in terms of functionality and also in terms of how approachable it is to the people who use it.

2.1.2 Impact of Software Architecture

Software architecture can play an important role in various aspects of software development. These include, but are not limited to [Gar00]:

Understanding Software architecture helps us understand software systems easily, exposing the high-level constraints on system design;

Reuse Architectural descriptions support reuse of component libraries, large components and the frameworks into which these components can be integrated;

Construction Architectural descriptions provide a partial blueprint for software systems by explicitly indicating components and dependencies between them (for example, an architecture's layered view usually displays abstraction boundaries between parts of a system's implementation, major internal system interfaces and the parts of systems that rely on other parts);

Evolution Software architecture can expose the various ways in which a software system is expected to evolve and, as such, make system maintainers have a better grasp of ramifications of changes, therefore being able to provide more accurate estimations of modification costs and make changes that address evolivng concerns regarding the system's overall performance;

Analysis Architecture descriptions facilitate analysis of software systems, providing new ways to analyze systems, such as:

- System consistency checking;

- Conformance to constraints imposed by an architectural style;
- Conformance to quality attributes;
- Dependency analysis;
- Domain-specific analysis for architectures built in specific styles;

Management The implementation of a software architecture tends to lead to a much clearer comprehension of requirements, implementation strategies and potential risks; leading to a more effective management of software systems from industrial and financial standpoints.

2.1.3 Development of Software Architecture

Beforehand, software architecture development was mostly an 'ad-hoc' affair, where descriptions relied on box-and-line diagrams that were rarely maintained once a system finished implementation. Architectural choices were devised in a rather idiosyncratic fashion and there was virtually no way to analyze an architecture for consistency or to check if a specific software system's performance accurately portrayed its architectural design.

As time passed, however, people started to realize the critical role that architectural design played on software systems and, as such, began pushing for a more detailed and disciplined approach on the subject. This gradually led to the rise of two trends centered around software architecture [Gar00]:

1. The recognition of a plethora of methods, techniques, patterns and idioms for structuring complex software systems;

2. The concern with exploiting similarities between specific domains to provide reusable architecture frameworks for product families, effectively giving software architectures longevity after their initial use.

All of this made the evolution from high-level programming languages to abstract data types and to software architecture development as we know it today possible [GS93], as more abstract factors relevant to the development of software architecture have been devised in a manner that made architectures more visible as a vital step in the software development process [Gar00].

2.1.3.1 Architecture Description Languages

Architecture Description Languages (***ADLs***) are notations responsible for characterizing software architectures, typically also providing tools for parsing and generating software architecture descriptions. The main contribution of ADLs is that they contribute to the decodification of a software system's design. [Gar00, MT10]

It is convenient to be able to distinguish ADLs from other programming languages. ADLs follow a classification and comparison framework described in Table 2.1 [MT00, MT10]:

ADL			
Architecture Modeling Features			
Components	**Connectors**	**Architectural Configurations**	**Tool Support**
Interface	Interface	Understandability	Active Specification
Types	Types	Composionality	Multiple Views
Semantics	Semantics	Hetereogeneity	Analysis
Constraints	Constraints	Constraints	Refinement
Evolution	Evolution	Refinement and Traceability	Code Generation
Non-Functional Properties	Non-Functional Properties	Scalibility	Dynamism
		Evolution	
		Dynamism	
		Non-Functional Properties	

Table 2.1: Classification and comparison framework for ADLs

Examples of ADLs include ACME, Adage, Aesop, C2, Darwin, Meta-H, Rapide, SADL, UniCon and Wright; each one of these having its own characteristics based on the framework above. For example, Table 2.2 is an accurate framework representation of the Darwin language [Gar00, MT00]:

Darwin			
Architecture Modeling Features			
Components (characteristics: component; implementation independent)	**Connectors (characteristics: binding; in-line; no explicit modeling of component interactions)**	**Architectural Configurations (characteristics: binding; in-line)**	**Tool Support**
Interface points are services (provided and required)	No interface (allows 'connection components')	Implicit textual specification which contains many connector details and provides graphical notation	Automated addition of ports to communicating components; propagation of changes across bound ports; dialogs to specify component properties
Extensible type system; supports parametrization	No types	Supported by language's composite component feature	Textual, graphical and hierarchical system view
Pi (3.14) calculus	No semantics	Allows multiple languages for modeling semantics of primitive components and supports development in C++ only	Parser; compiler; 'what if' scenarios by instantiating parameters and dynamic components
Language via interfaces and semantics	No constraints	Provided services may only be bound to required services and vice-versa	Compiler; primitive components are implemented in a traditional programming language
No evolution	No evolution	Supports system generation when implementation constraining	Compiler generates C++ code
No non-functional properties	No non-functional properties	Scalability hampered by in-line configurations	Compilation and runtime support for constrained dynamic change of architectures (replication and conditional configuration)
		Evolution hampered by in-line configurations; no support for partial architectures or application families	
		Constrained dynamism: runtime replication of components and conditional configuration	
		No non-functional properties	

Table 2.2: Darwin's Framework Results

It is also worth mentioning that, despite not originally being considered an ADL, the UML is not only used to document software architectures (in a similar fashion to all the ADLs described above), but it is also considered the standard notation for the practice, effectively turning it into a legitimate ADL [Pan10, MT10].

2.1.3.2 Architectural Styles

Architectural styles (also referred to as models or patterns) are the elements in the field of software architecture that typically have the role of specifying a design vocabulary, restrictions on how that vocabulary is employed and semantic assumptions regarding that vocabulary [PW92, Gar00, MT10].

There are various, distinct examples of architectural styles. Table 2.3 shows a complete list [SKA15]:

Application Type	**Architectural Styles**
For applications focused on shared memory	Repository, data-centric, rule-based
For distributed systems	Client-server, space based architecture, peer-to-peer, shared nothing architecture, broker, representational state transfer, service-oriented
For applications focused on messaging	Event-based, asynchronous messaging, publish-subscribe
For applications focused on structure	Object-based, pipe-and-filter, monolithic-application-based, layered architecture
For adaptable systems	Plug-ins, reflections, microkernel
For modern systems	Grid computing, multi-tenancy, big-data

Table 2.3: Complete list of architectural styles

Out of the architectural styles mentioned above, it is required to have a good idea as to how the most important ones work [SKA15, GS93, Gar00]:

Pipe-and-filter Each component contains a set of inputs (where data streams are read) and outputs (where data streams are produced). The 'pipes' are the connectors that provide the path for the data streams to navigate from one component to another. The 'filters' are the components themselves, which transform input streams locally in a way that allows the production of output streams to begin before the input streams are even received (therefore making the entire process more effective). Threads and processes are accurate examples of filters.

This style is illustrated in Figure 2.1:

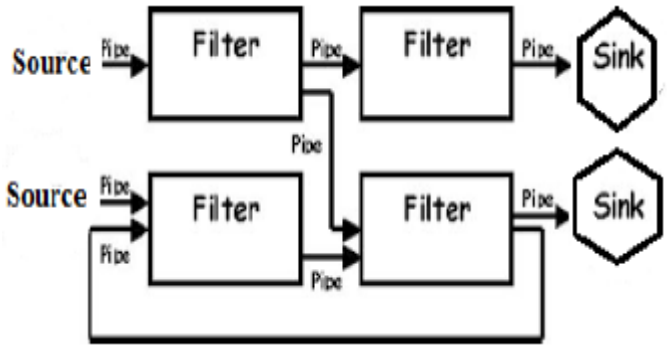


Figure 2.1: Pipe-and-Filter Architecture

Object-based (or object-oriented) The components are objects that interact through function and procedure invocations. This style is based on the principle of separation of concern, where a system is divided into several partitions and each partition deals with its own separate concern. As such, in this style, each object possesses the function of preserving the integrity of its representation, which is kept hidden from other objects. The components in this particular model can be divided into:

Fully or partially experienced components Components belonging to the main organization's library, used in developing several systems;

Off-the-shelf components Components that belong to third-party libraries;

New components Components built from scratch (and are therefore not available in any library).

This style is illustrated in Figure 2.2:

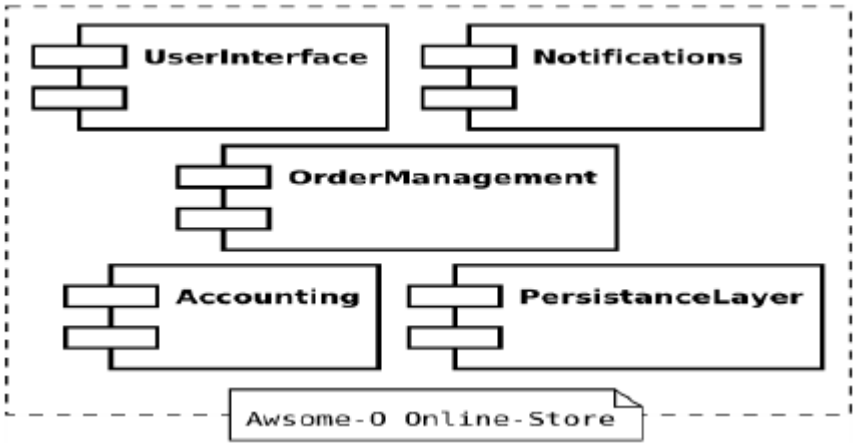


Figure 2.2: Object-Based Architecture

Event-based (or implicit invocation) Instead of invoking a procedure directly, a component can also invoke it by broadcasting one or more events, where other components can associate a

procedure to each event and the system itself then invokes all the procedures that have been registered for the event(s). The event-based model is divided into four layers:

Event generator Responsible for the generation of events (as a result of an action such as a mouse click or the press of a keyboard button);

Event channel Responsible for transferring an event from its source to the event processing engine, shortly after being placed in a queue;

Event processing engine Where events are processed and appropriate responses to the events are generated;

Downstream event-driven activity Where consequences of events are shown.

This style is illustrated in Figure 2.3:

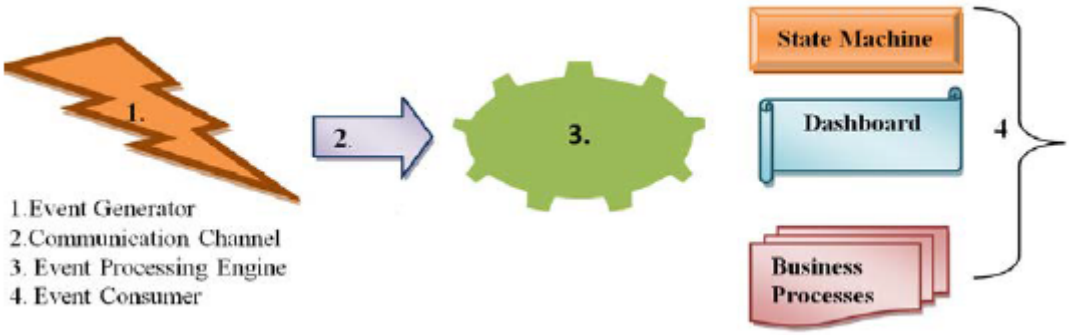


Figure 2.3: Event-Based Architecture

Client-server Distributed model where the server process (examples of servers being a web server, a database server and a FTP server, among others) provides services to the client processes (examples of clients being a browser, an online chat client and an e-mail client, between others) according to the clients' requests. If it is required, a client can become a server (or vice-versa) and/or a server can provide services with the help of other servers. In most instances, the server does not know the number of clients that will access it, or their identities, in advance. Clients, on the other hand, have the ability to know a server's identity (often by consulting other servers) and access it via remote procedure call. There are two types of the client-server architecture model:

2-Tier Client and server interact without any point or node in between;

3-Tier Client and server interact with a node in between, the purpose of the node being to authenticate and approve requests from the client and then pass approved requests to the server, acting as a verifier and a validator in the client-server model.

This style is illustrated in Figure 2.4:

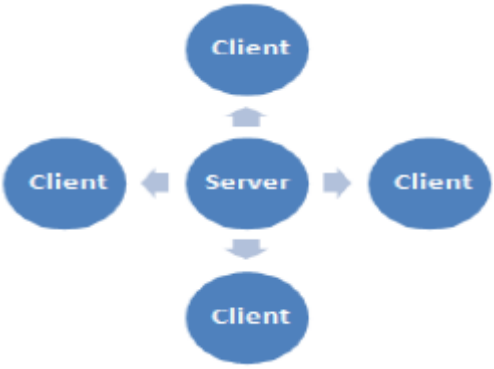


Figure 2.4: Client-Server Architecture

Layered architecture Hierarchically organized model divided by layers based on the principle of separation of concern. Each layer can communicate with other layers through well-groomed interfaces. Commonly, a software system that uses this architectural style is divided into three layers:

Presentation Layer Layer that provides user interface functionalities;

Business Layer Layer that implements the system's business logic;

Infrastructure Layer Layer that incorporates infrastructure services (such as services relevant to data and networking).

This style is illustrated in Figure 2.5:

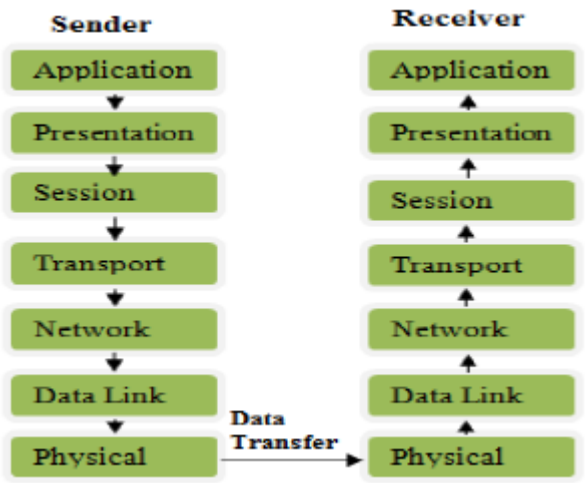


Figure 2.5: Layered Architecture (OSI Model)

Repository Style represented by a central data structure representing its current state and external components that operate independently on said structure as knowledge sources trying to solve a given problem. Depending on what the main trigger of selection of processes to execute is, there are two subcategories of repositories:

1. If the selection is triggered by the types of transactions in an input stream of transactions, then it is a **traditional database**;
2. If the selection is triggered by the current state of the central data structure, then it is a **blackboard**.

This style is illustrated in Figure 2.6:

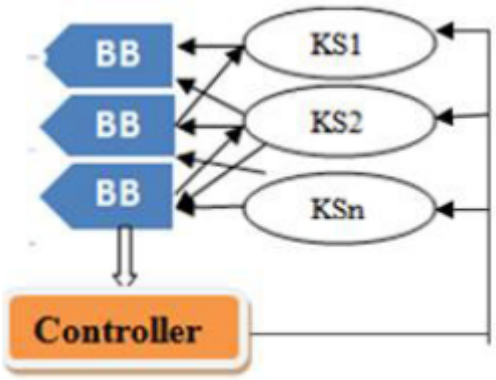


Figure 2.6: Blackboard Architecture

2.1.3.3 Product Lines and Standards

The ongoing desire to make good use of the commonalities between software systems has led to the rise of the product-line approach and the cross-vendor integration standards.

The product-line approach to software architecture, illustrated in Figure 2.7, directly involves building an architecture that supports a line of software systems, as opposed to a single solitary system (that would be the single-product approach). Organizations such as the SEI (Software Engineering Institute) have already begun to provide guidelines on how to implement the product-line approach [Gar00].

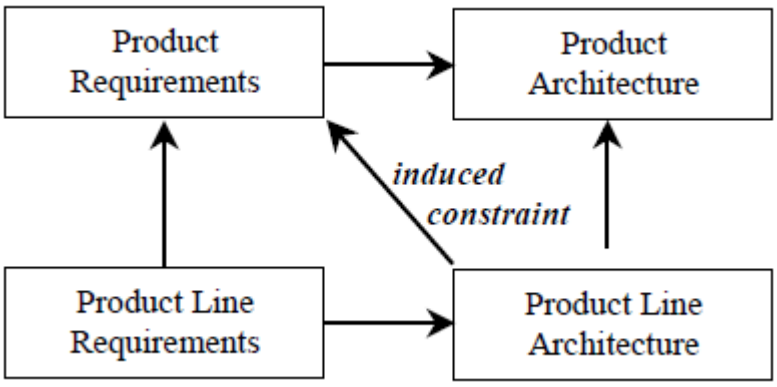


Figure 2.7: The Foundation of Product Line Architectures

On a similar note, cross-vendor integration standards typically allow the system to support integration of parts provided by multiple vendors. Depending on the range of parts compatible

with the system, such standards may be international standards (such as the ones sponsored by the IEEE and the ISO) or ad-hoc standards defined by the leader(s) of a specific industry [Gar00].

2.1.3.4 Software Architecture Viewpoint Models

Software architecture is organized into different views. In this context, a view is understood as a set of models representing a system from the perspective of a related set of functional or non-functional requirements (also known as 'concerns') raised by stakeholders. For the purposes of creating, depicting and analyzing views, we have viewpoints and viewpoint models [May05].

Examples of viewpoint models are [May05, MT10]:

Kruchten's '4+1' View Model Model consisted of multiple views that allow their respective concerns to be dealt with separately. These views, as shown in Figure 2.8, are:

Scenarios View Where the critical scenarios are conceived and described;

Logical View Where key abstractions from the problem domain are identified and modeled, forming logical classes;

Development View Where logical classes are mapped to modules and packages;

Process View Where logical classes are mapped to tasks and processes;

Physical View Where the processes and modules are mapped to the hardware.

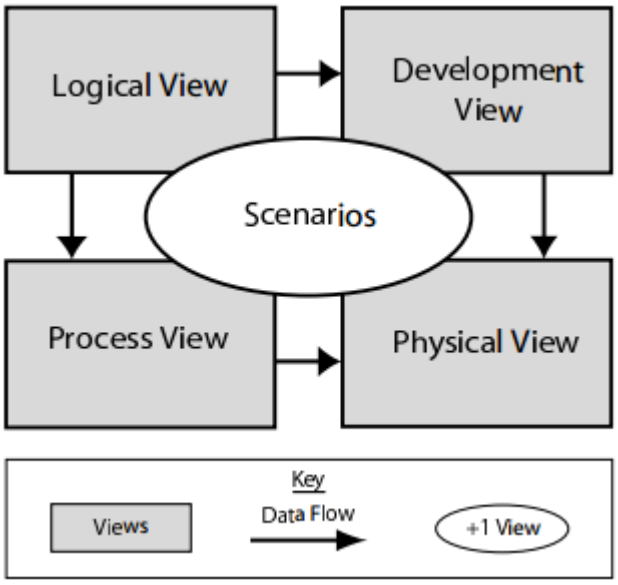


Figure 2.8: Kruchten's '4+1' View Model

SEI Viewpoint Model Model that is based on a list of styles and views founded upon the structure of the system itself that can be used for the representation of said system. These styles are related to each other through their viewtypes (module, component-and-connector (***C&C***) and allocation), which don't interact much between one another (except for the allocation

viewtype). To reduce this model's complexity, styles that are less relevant to stakeholder concerns can be omitted. This model also includes the template for the contents of a viewpacket, which is a fraction of a view that is too complex for a single representation.

ISO Reference Model of Open Distributed Processing Model that provides a framework to develop standards for the distribution of information processing services. This model's specification is divided into five viewpoints: enterprise, information, computation, engineering and technology. These viewpoints are simplified through the model's specification of transparencies. There are no specific stakeholders for any of the RM-ODP model's views, but it is designed to meet the needs of system developers in an abstract manner. The model also does not specify any links between views, although it does provide translation rules to help define the relationships between elements of different views.

Siemens Four View Model A result of a study into the industrial practices of software architecture, this model's views are based on the four broad categories that the structures used to design and document software architecture fell into. A good number of important mappings of structures are openly defined in the design approach. This model's loosely coupled views, as shown in Figure 2.9, are:

Conceptual View View that represents the beginning of the model's design flow;

Module View View where conceptual structures are implemented;

Code View View where module structures are implemented and execution structures are configured;

Execution View View that the module structures are assigned to.

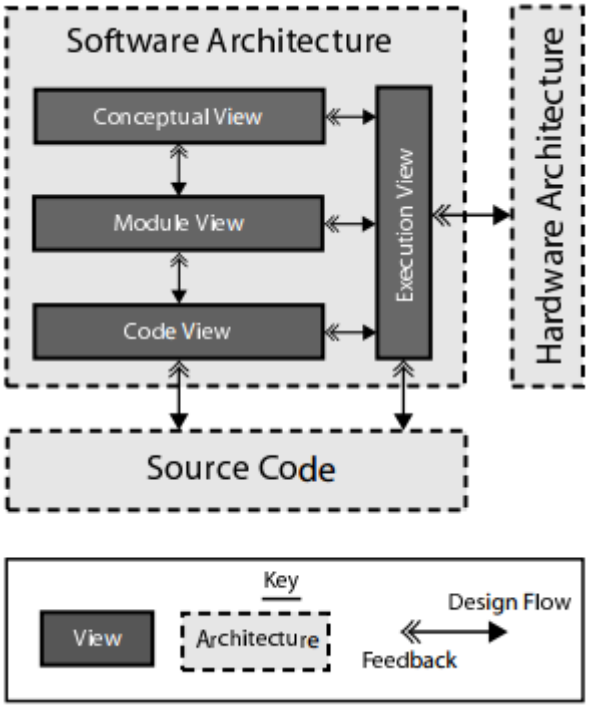


Figure 2.9: Siemens Four View Model

Rational Architectural Description Specification (ADS) An expansion on Kruchten's '4+1' model to enable the description of larger and more complex architectures, shown in figure 2.10. The model is defined as such:

- Requirements Viewpoint
  - Domain View
  - Use Case View (formerly the Scenarios View)
  - User Experience View
  - Non-Functional Requirements View (loosely coupled with the other views, therefore being omitted when discussing this model's concerns)
- Design Viewpoint
  - Logical View
  - Process View
- Realization Viewpoint
  - Implementation View (formerly the Development View)
  - Deployment View (formerly the Physical View)
- Verification Viewpoint
  - Test View

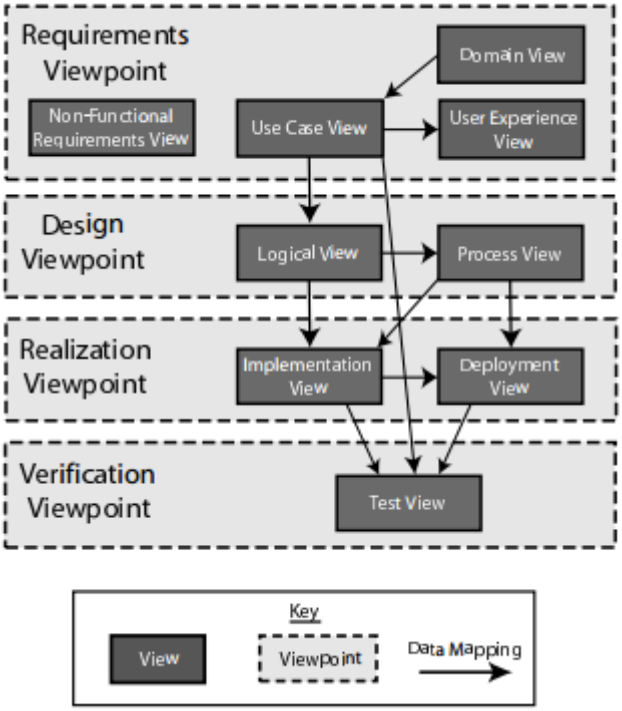


Figure 2.10: Rational ADS Model

2.1.3.5 Connections with Agile Software Development

Various experts in the field have been trying to discover the role of software architecture in agile software development approaches. The main problem here is that the two fields might not converge in an appropriate fashion. Advocates of software architecture as a key tactic for quality systems doubt the efficiency of any development approach that does not place enough focus on architecture, while proponents of agile development argue that software architecture is outdated and has little end value from a customer perspective [ABK10].

How can a reconciliation point between these two fields be found? The answer might lie in solving the issues that fuel the conflict between them [ABK10]:

Semantics When this thesis addressed the Definition of "Software Architecture", it helped clarify the semantics around software architecture. This was an essential affair, as the concept tends to have fuzzy boundaries, the main one being that, despite the fact that there is a lot of overlap between design decisions and architectural decisions, not all design is architecture.

Context The amount of architectural activity needed varies from project to project. In each specific scenario, attention must be paid to influencers such as project size, business model, team distribution, market situation and the company's power and policies. If the project requires an accentuated architectural effort, the agile development methods will have to be adjusted to suit that effort.

Life Cycle When, in a project's life cycle, should we start focusing on software architecture? Given that a software system's architecture represents significant decisions regarding said system's structure and behavior (and therefore will later be considered the hardest to undo or refactor), preferably as early as possible.

Roles When it comes to software architecture, who exactly are the 'architects'? For larger and more demanding systems, the presence of two specific roles will be key:

- The *architectus reloadus*, maker and conceiver of big decisions, responsible for the project's external coordination;
- The *architectus oryzus*, mentor and troubleshooter, responsible for the project's internal coordination.

Documentation The amount of documentation the architecture of each system needs will depend on that system's scope and simplicity. If the system is easy, short and simple, then an architectural prototype with limited documentation will suffice. However, if the system is tricky, large and complex, then that will require more detailed documentation.

Methods Software architecture's efficiency is proved by the methods that it uses to solve architectural issues, which are issues that agile development is silent on and, at the same time, affected by. This thesis has already covered some of those methods, such as Architecture Description Languages and Architectural Styles.

Value Agile approaches strive to deliver business value early and often. The issue here is that, while the cost of a software architecture is visible, the architecture's value is not. As such, an approach that manages to find the right compromise between architecture and business value is needed (a suggested example being incremental funding).

2.1.3.6 Other Factors of Software Architecture Development

It is more than evident by now that software architecture development is a very expansive subfield. There are still other factors that did not get previously discussed, including, but not limited to:

Software Architecture Erosion Control Methods to try to control software architecture erosion, which is understood as a software architecture's deviation from its original design principles with incremental changes, therefore gradually making that architecture no longer useful to its intended purpose nor economically viable to maintain [SB11].

Software Architecture Recovery The set of methods that seek to extract higher levels of abstraction from existing software systems and/or architectures by recovering the past design decisions that had been taken by experts and then lost due to a multitude of possible reasons (for example, document revisions or assumptions that were initially disregarded) [RA07].

2.1.4 Examples of Software Architectures

Some examples of software architectures include:

International Standard Organization's Open Systems Interconnection Reference Model Layered network architecture [GS93];

NIST/ECMA Reference Model Generic software engineering environment architecture based on layered communication substrates [GS93];

X Window System (X11) Distributed windowed user interface architecture based on event triggering and callbacks [GS93];

High Level Architecture for Distributed Simulation Architecture that allows the integration of simulations (which work as the architecture's 'applications') provided by vendors and operates based on international cross-vendor integration standards [Gar00];

Java's Enterprise Beans Architecture Architecture that supports distributed, Java-based, enterprise-level applications and operates based on ad-hoc cross-vendor integration standards defined by Java's industrial leaders (in other words, the standards are defined by Java for vendors to provide parts that exclusively support Java applications) [Gar00].

To give an example of a visual representation of one of those systems, Figure 2.11 offers an illustration of the X Window System:

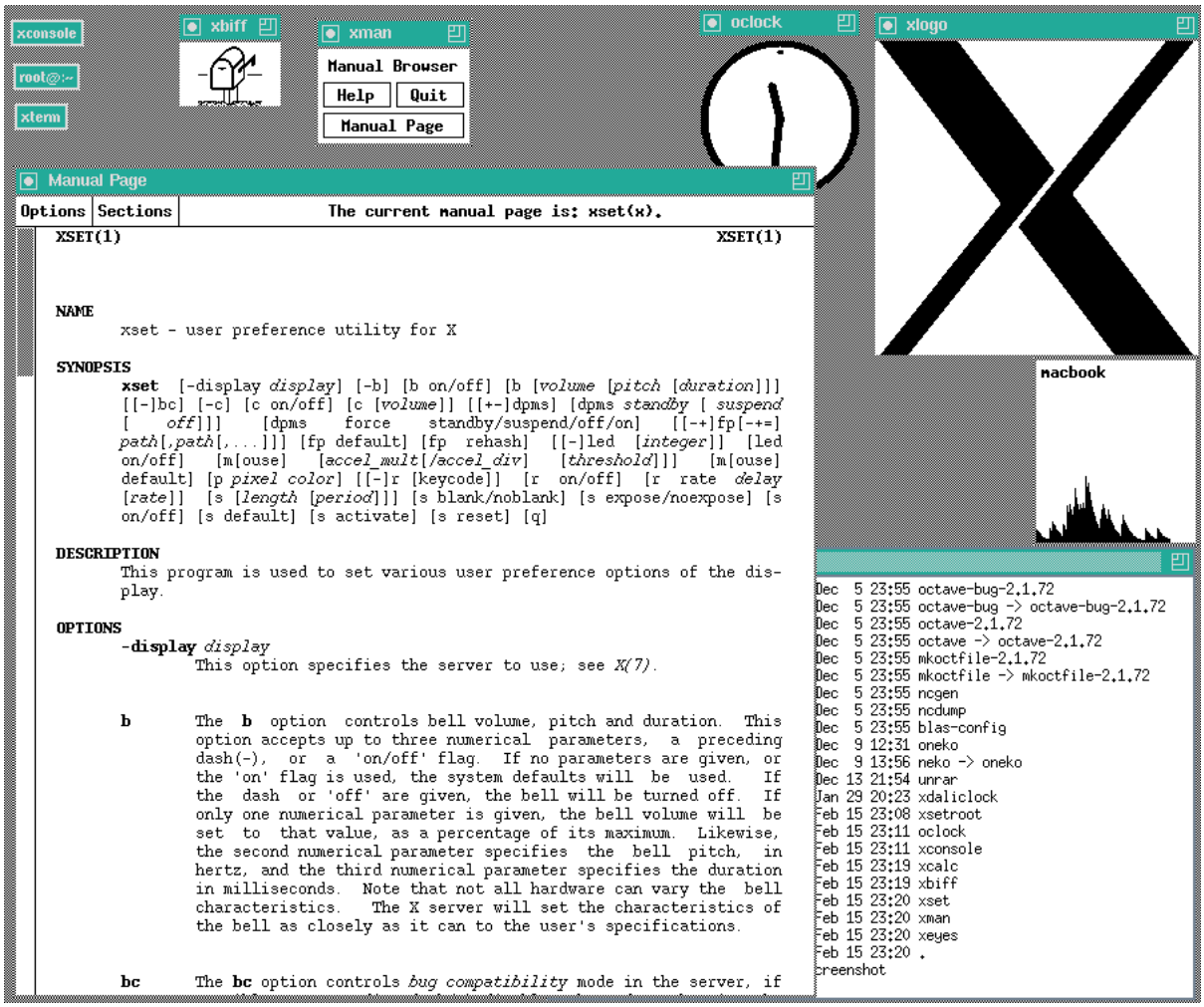


Figure 2.11: An Example of the X Window System in Execution (Liberal Classic, 2007)

Software architectures can also have succint descriptions that inform users about some of their inner details. Here are some examples of those descriptions, each one of them being relevant to a specific software architecture [GS93]:

- *Camelot is based on the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and servers;*
- *Abstraction layering and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a client-server model for the structuring of applications;*
- *We have chosen a distributed, object-oriented approach to managing information;*
- *The easiest way to make the canonical sequential compiler into a concurrent compiler is to pipeline the execution of the compiler phases over a number of processors.* [...] *A more*

> *effective way* [is to] *split the source code into many segments, which are concurrently processed through the various phases of compilation* [by multiple compiler processes] *before a final, merging pass recombines the object code into a single program.*

From these descriptions, as well as everything that has been covered thus far regarding this topic, we can conclude that different software architectures can function in a different manner from one another, which contributes to software architecture's overall versatility.

2.2 Software Design

When we think of the field of SAD, software architecture is the topic that is the most important to keep in mind. However, that does not mean that software design is completely irrelevant, either. Agile software development was already covered earlier when we described the Connections with Agile Software Development, but another notable example of software design in the context of SAD in action is the SOLID mnemonic.

2.2.1 The SOLID Mnemonic

The SOLID mnemonic describes a set of principles for object-oriented software design, which is specially relevant to the serious game [Mar02]:

S The Single Responsibility Principle, which states that a class should only have one reason to change, meaning that a class should only have one job;

O The Open-Closed Principle, which states that objects or entities should be open for extension, but closed for modification;

L The Liskov Substitution Principle, which states that, if T is a type, S is a subtype of T and f(x) is a property provable for objects x of the type T, then f(y) should be provable for objects y of the type S;

I The Interface Segregation Principle, which states that a client should never be forced to implement an interface it does not use and that clients should not be forced to depend on methods they do not use;

D The Dependency Inversion Principle, which states that entities should not depend on low-level modules (concretions), but on high-level modules (abstractions).

2.3 Serious Games

When all is said and done, the only factor that sets a serious game apart from your average game is the former's focus on serious topics. Outside of that, the two types of games may use similar game design philosophies and even operate in a similar manner.

As such, it is recommended that regular game design rules are kept in mind while developing serious games.

2.3.1 Game Design Rules

Applying game design rules to a serious game means that game not only needs to successfully teach the player about the topic it revolves around, but it also needs to [Coe15]:

- Give the players an effective journey - get them into the game, give them a natural sense of progression and provide them mastery;
- Give the players a consistent and fair difficulty balance that lacks stagnation and forced or trivial decisions;
- Create a compelling experience, making the game a scenario of another reality that captures the players' focus;
- Give the players a convincing motivation to keep playing the game - either through the points-badges-leaderboards triad or another method that relies on the game's components, depending on the situation.

A system that is particularly worth keeping in mind while making a serious game is the Pyramid of Elements, devised by Kevin Werbach and illustrated in Figure 2.12. The different levels of this pyramid are, from highest to lowest [Coe15]:

Dynamics Consisted of elements that are integral to the feel of the game, such as...

- Constraints
- Emotions
- Narrative
- Progression
- Relationships

Mechanics Consisted of elements that relate to the game's more technical aspects, such as...

- Challenges
- Chance
- Competition
- Cooperation
- Feedback
- Resource acquisition
- Rewards

- Transactions
- Turns
- Win states

Components Consisted of the more concrete objects within the game, such as...

- Achievements
- Avatars
- Badges
- Boss fights
- Collections
- Combats
- Content unlocking
- Gifting
- Leaderboards
- Levels
- Points
- Quests
- Social graph
- Teams
- Virtual goods

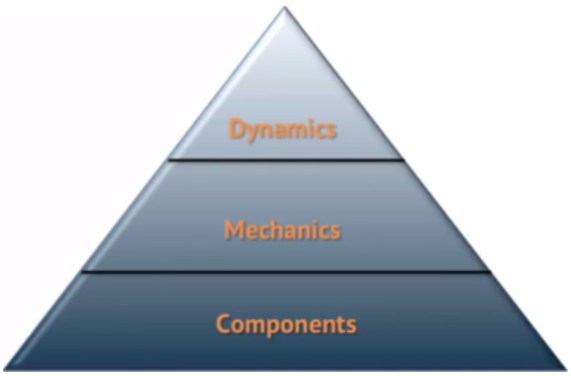


Figure 2.12: The Pyramid of Elements (Kevin Werbach)

2.3.2 Serious Game Design Patterns

The optimal method to design an effective educational game for an SE topic is to utilize learning and teaching functions (***LTFs***) [Gro07] that are convertible to game design patterns (***GDPs***) [BH04], more precisely the subset of LTFs specific for SE topics, which translate to specific GDPs

relevant to those topics. This dynamic between learning and teaching functions, game design patterns and the serious game is known as the LTFs-GDPs-SG Triangle, illustrated in Figure 2.13.

It is convenient to know what those LTFs and GDPs are. Table 2.4 represents a list of LTFs relevant to SE, while Table 2.5 represents a list of GDPs relevant to SE [LPF15].

Preparation	Prior knowledge activation, motivation, expectations, attention
Knowledge Manipulation	Encoding, comparison, repetition, interpreting, exemplifying
Higher Order Relationships	Combination, integration, synthesis, classifying, summarizing, analyzing
Learner Regulation	Feedback, evaluation, monitoring, planning
Productive Actions	Hypothesis generation, inferring, explaining, applying, producing and constructing

Table 2.4: Learning and Teaching Functions (SE)

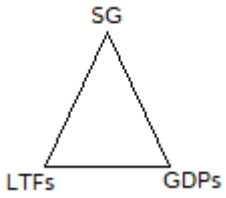


Figure 2.13: The LTFs-GDPs-SG Triangle

Pattern Type	**Description**	**Number of Patterns**	**Example of a Pattern**
Game Elements Patterns	Game objects that define the area of the game reality or that players can manipulate	48	Clues
Patterns for Resources and Resource Management	Different types of resources that can be controlled by the players or the game system	20	Energy
Patterns for Information, Communication and Presentation	Game information and how it is treated	20	Asymmetrical Information
Actions and Events Patterns	Actions that are available to the players and how they relate to game state changes or players' goals	44	Rewards or Penalties
Patterns for Narrative Structures, Predictability and Immersion	Story line, immersion and commitment to the game by the players	31	Surprises
Patterns for Social Interaction	Social interaction between the players	30	Role-playing
Patterns for Goals	Objectives to give to the players while they are playing games	26	Gain of Information
Patterns for Goal Structures	Methods in which gameplay affects goals	20	Tournaments
Patterns for Games Sessions	Characteristics of games instances and game and play sessions and the limitation, possibilities and features of player participation in the game	20	Time Limits
Patterns for Game Mastering and Balancing	Methods in which the players hone their skills and abilities while playing the game and the gameplay itself becomes balanced for players with different abilities	27	Randomness
Patterns for Meta Games Replayability and Learning Curves	Factors outside the playing of a single instance of the game	10	Replayability

Table 2.5: Game Design Patterns (SE)

2.3.3 Examples of Serious Games

There is a sizable number of serious, educational games that have been developed for SE topics. A survey of existing SGs for SE topics was performed, where a set of games, shown in Table 2.6, was identified. Those games are divided by the SE subtopic they address [Far16, Cru10]:

Process/Project Management	*SimSE* [N+10], *SE RPG* [Ben], *PlayScrum* [FS10], *SESAM* [kn:b], *Sim.js* [Var11], *X-MED* [kn:a], *E4* [Cru10]
Requirements	*Ilha dos Requisitos* [TZG10]
Testing	*U-Test* [U+], *iTest Learning* [FMCS12], *iLearnTest* [Rib14]

Table 2.6: Examples of Serious Games

It is convenient to know about some of those games more in detail [Far16]:

SimSE Serious simulation game about software process and project management. In it, the player acts as a project manager of a team of developers working on a certain number of SE projects, with the goal of successfully finishing them. Some of the actions the player can do include hiring or firing employees, assigning tasks to them, monitoring their progress, and purchasing tools. The graphics and user interface of the game can both be seen in Figure 2.14, displaying the virtual working environment and information about artifacts (size, completeness, correctness), customers (overall satisfaction) and tools (number of users, productivity increase factor), among other things [N+10].

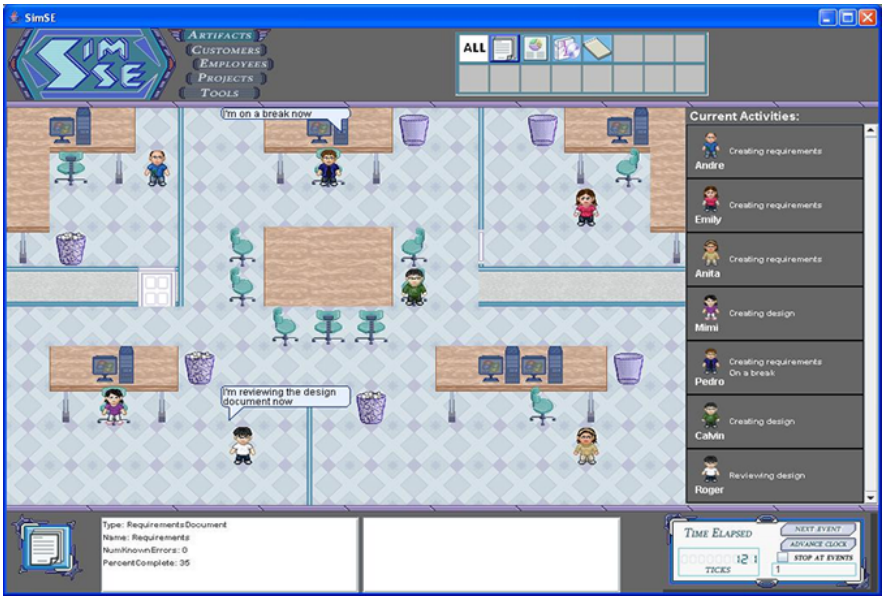


Figure 2.14: *SimSE* (E. Navarro, University of California, 2010)

Ilha dos Requisitos Serious online adventure game about requirements in software, illustrated in Figure 2.15. It is about an amateur system analyst who finds himself in an isolated island after a plane accident and must find a way to escape it (along with a resident tribe) before the island's volcano erupts. In order to succeed, the player must clear a number of challenges that involve software requirements [TZG10].

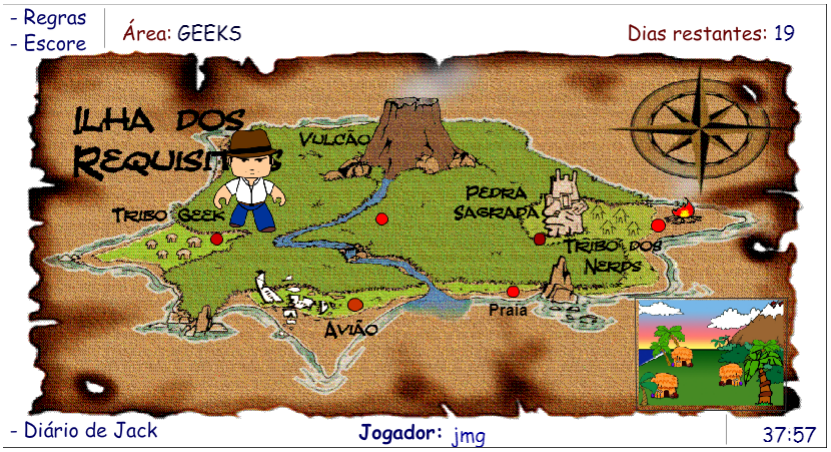


Figure 2.15: *Ilha dos Requisitos* (M. Thiry, UNIVALI, 2010)

iLearnTest Serious 2D platformer game about software testing, illustrated in Figure 2.16. The player is informed about what chapter each colored platform represents, as well as the player's current score in each chapter (as compared to the maximum possible score). When the player goes 'inside' the colored platforms, they get learning objectives related to software testing, followed by different types of challenges involving those learning objectives. Those challenges can range from, but aren't limited to, drag-and-drop challenges to find-the-right-path challenges [Rib14].

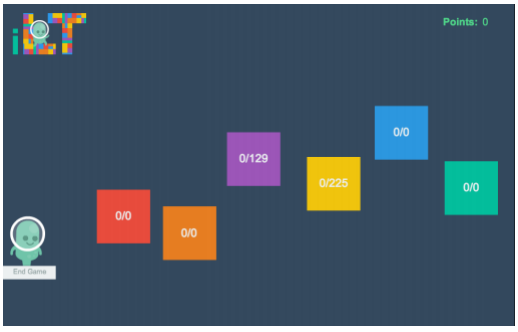


Figure 2.16: *iLearnTest* (T. Ribeiro, FEUP, 2014)

2.4 Summary

Software engineering is a field of increasing interest. One of the field's subgenres is Software Architecture and Design.

Software architecture is the specification of the structure of a software system, and it carries a net positive effect on the software systems it is implemented in. Software architecture development is very expansive, with a considerable number of essential factors surrounding it. Different software architectures can also operate in a different manner.

Although it does not play a major role in the field of SAD, software design is still worthy of note.

Serious games often operate by the same game design rules as regular games, and they tend to follow various game design patterns. A noticeable number of examples of serious games for SE can be found.

Chapter 3

Serious Game for Learning About Software Architecture and Design

There is no doubt that traditional software engineering education models employed by educational institutions have been efficient at teaching people about various software engineering topics. However, there are also some problems with these models that are worth addressing.

In particular, these models can sometimes be hard to put in practice, less efficient than self-paced learning, and/or have information that is too complex for the students to effectively absorb. Furthermore, these models can also be established in a way that fails to motivate the students to pay attention to what is being taught. The interest in building a serious game is that it manages to effectively educate students while simultaneously solving (or at least minimizing) these issues.

In page 25, various examples of serious games were addressed, each one of them focusing on a certain SE topic. However, there have not been any educational games found for our topic of interest, which is Software Architecture and Design (at least when it comes to digital games).

Since there are no digital games based around Software Architecture and Design, it is yet to be determined whether the GDPs particular to SE topics (previously discussed in Serious Game Design Patterns) function for that specific topic. In order to solve that problem, we proved that premise to be true by using the GDPs relative to SE topics to design and implement a serious game for the SAD topic, and then validated that game using an ESWS that tested how much of the topic the test subjects managed to learn through the game itself.

Before talking about that study, it is necessary to describe the serious game in question. It is a 2D platformer called ***Codebase Escape***, developed using the Unity game engine and the C# scripting language. It has versions for Windows, Mac and Linux, respectively. In terms of core functionalities, the game works like the prototypical 2D platformer, where the main character can navigate by moving left or right and jumping from platform to platform. The game also includes outsourced graphics and audio. Figure 3.1 shows the game's logo.

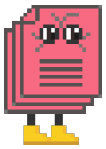
Figure 3.1: Codebase Escape Logo

The plot of the game is as follows: an anthropomorphized piece of source code keeps increasing itself while trying not to become a mess - in other words, a 'big ball of mud', or ***BBOM*** for short.

The game follows a 'butterfly' analogy, where the code keeps maturing so that it can 'escape the codebase' and, in the process, become a fully realized piece of code that can be used as the 'fuel' of a program (or possibly more programs), in a similar fashion to a butterfly trying to escape its cocoon. The three stages the main character goes through as the game continues is shown in Figure 3.2. The player also takes the role of the 'programmer', essentially controlling the code, making it grow and leading it to the end.

(a) Initial Version

(b) Intermediate Version



(c) Matured Version

Figure 3.2: The Main Character

The game's default controls are shown in Table 3.1a:

Action	**Keyboard Controls**	**Joystick Controls** (a)
Moving left	A or left arrow	Joystick horizontal axis (left)
Moving right	D or right arrow	Joystick horizontal axis (right)
Jumping	W, space or up arrow	Joystick vertical axis (up)
Performing a specific context-dependent action	Left Ctrl or mouse left-click	Joystick button 2
Checking inventory	Left Alt or mouse right-click	None
Pausing	Enter	Joystick button 0
Restarting the level (only works in certain scenarios)	Left Shift or mouse wheel click	Joystick button 3
Exiting the game	Esc	Joystick button 1

Table 3.1: Default Game Controls

(a) Specific joystick controls depend on the joystick being used.

When the player starts the game, they will be taken to a 'Configuration' window, where they can configure the game's graphics and controls (although it is recommended that they do not alter the custom resolution on each version).

3.1 Success State

There are two types of items that are scattered around each of the 5 levels and the player will need to pay attention to in order to advance. These items are:

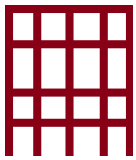
Lines of code (LOCs) Pieces of code that the player collects in order to unlock the quiz at the end of each level (example shown in Figure 3.3). There are 5 LOCs in Level 1, 6 in Level 2, 7 in Level 3, 8 in Level 4 and 1 in Level 5;

Documentation files (DOCs) Pieces of information around the topic of SAD that the player will need to consult in order to solve each quiz (example shown in Figure 3.4). The first level has only one DOC while all the other levels have two.

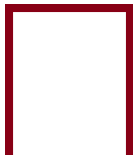
Figure 3.3: Line of Code

Figure 3.4: Documentation File

In this game, each quiz is shown as a door that takes the player to the quiz. Before collecting all the LOCs in a level, the door is locked; and after collecting all those LOCs, the door is unlocked. Both the locked and unlocked versions of the door can be seen in Figure 3.5:



(a) Locked Quiz Door



(b) Unlocked Quiz Door

Figure 3.5: Quiz Door

After the player solves the quiz, the level is cleared. If the player clears all 5 levels, they clear the game.

Before and after each level there is a transitional scene, known as a 'tome of knowledge', which includes additional information that is not essential to solve any of the quizzes, but is relevant to the empirical study with students this game was used in.

The game also has a scoring system, which works as such:

- The score starts at 5000 and keeps decreasing as the player plays the level;
- The score decreases in a slower rate when the player either reads a DOC or attempts to solve a quiz, which is done to help players focus while simultaneously testing their mental agility;
- The score stops decreasing when it gets to 0;
- Successfully clearing a level grants the player with a score boost, which gets higher as the player becomes increasingly closer to the end:
  - Clearing level 1 will lead the player to start level 2 with the score they ended level 1 with plus 5000;

  - Clearing level 2 will lead the player to start level 3 with the score they ended level 2 with plus 10000;
  - Clearing level 3 will lead the player to start level 4 with the score they ended level 3 with plus 15000;
  - Clearing level 4 will lead the player to start level 5 with the score they ended level 4 with plus 20000;
  - Clearing level 5 will lead the player to finish the game with the score they ended level 5 with plus 50000;

- This all means that the player scores higher the faster the player can clear each level (making 50000 the minimum score the player can end the game with).

As an added bonus to incentivize players not to guess the correct answer by trial and error (without reading the DOCs), players also get an extra health point (***HP***) if they answer the quiz correctly in their first attempt. The HP count is represented by a health image, with a number on top that represents the amount of HPs the main character has left at each point in the game. The HP is illustrated in Figure 3.6:

Figure 3.6: Health Point

3.2 Failure State

The game will also have enemies that patrol certain sections of levels, most of them being illustrated in Figure 3.7. They act as so-called 'code smells', which are parts of code that are greatly problematic to the software's organization. As such, will try to turn the main character into a BBOM.

The game has what is called 'interacting damage', where touching an enemy results in a certain negative consequence to the player's character. There are five types of enemies throughout the game. Here is what they are and do:

Duplicate Foe 1 Foe with a red 1 marked on it. For every time the main character hits it, it loses movement speed;

Duplicate Foe 2 Foe with a blue 2 marked on it. For every time the main character hits it, it loses jump power (and therefore cannot jump as high as it previously could);

Duplicate Foe 3 Foe with a purple 3 marked on it. It combines the effects of the Duplicate Foes 1 and 2 (which is referenced by the fact that the color of the marked number is a combination of red and blue). All three of the Duplicate Foes are named in reference to duplicate code, which is a well-known code smell;

Damage Foe Foe with a green 4 marked on it. For every time the main character hits it, it loses a HP;

Ultimate Foe The game's most dangerous foe, which also can be seen as the game's 'boss'. Unlike the previous four foes, where their look was designed based on the smell of painting instruments, this foe looks like a locked door, in a deliberate attempt to mislead the player into thinking they're about to clear the level. This foe is also initially static, and once the main character catches Level 5's doc, this foe will immediately proceed to pursue the main character, forcing the player to act and think fast for the rest of the level. If the main character hits this foe, it will lose an accentuated amount of movement speed and jump power and will also lose 5 HPs.

If all HPs are lost, the game is over, and afterwards the player can choose to either start a new game or return to the main menu.

(a) Duplicate Foe 1

(b) Duplicate Foe 2

(c) Duplicate Foe 3

(d) Damage Foe

Figure 3.7: Foes

The game also has a 'soft failure state' of sorts, where the main character turns into a BBOM. What this means is that the main character gets too much of his movement speed or jump power

taken away by foes, to the point where it can no longer proceed through the level in an adequate fashion. In this case (or if the player cannot proceed due to a bug), it is recommended that the player restarts the level. In order to do this, the players will need to pause the game and click the restart button, in that order (although this does not reset the score).

3.3 Game Progression

As the player keeps playing the game, two things should be kept in mind:

1. The levels get trickier and more complex as the game goes on;
2. The player's character gets heavier for each LOC it obtains, which means that its movement speed and jump height become lower.

As such, it is necessary to compensate for the increased weight in such a way that allows the player to clear each of the levels. In order to fulfill that condition, each quiz plays the part of a "code wizard" of sorts, essentially refactoring the player's character (which translates into giving the main character ability buffs) in a way that will result in a more well-organized piece of code. Also, as the code keeps growing, the effect the LOCs have on it decreases (although this does not extend to the effect the foes have on it). All of this means that the code gets 'stronger' and 'smarter' as the game continues.

Each quiz represents a principle within the SOLID set of principles, discussed in page 20. The outcome of passing each quiz is as follows:

- Clearing the quiz from the first level (*S*) will give the main character a slight increase in jump height, movement speed and current number of HPs, respectively;
- Clearing the quiz from the second level (*O*) will give it a significant increase in jump height;
- Clearing the quiz from the third level (*L*) will give it a significant increase in movement speed;
- Clearing the quiz from the fourth level (*I*) will give it a significant increase in current HP number;
- Clearing the quiz from the fifth level (*D*) will clear the game itself.

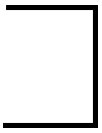
3.4 Other Important Details

At a specific point of the game, switches are introduced. Once activated by the main character, switches activate moving platforms that can be used by the player to reach previously unreachable platforms. For the most part, the player cannot tell for sure which platforms (or parts of platforms) will start moving once a certain switch is pressed. An example of a switch is presented by Figure 3.8:

Figure 3.8: Switch

At the last level, there will be chains of foes where the foes in question respawn at the starting point, move in direction to the ending point and then disappear at the ending point. Each of these two points is represented by the so-called 'chain limits'. If the enemies move from the left to the right, the left chain limit is the starting point and the right chain limit is the ending point, while if the enemies move from the right to the left, the right chain limit is the starting point and the left chain limit is the ending point. Examples of both the left and right chain limits are presented by Figure 3.9:

(a) Left Chain Limit



(b) Right Chain Limit

Figure 3.9: Chain Limits

The players may also view their inventory by pressing the inventory button. At each point a player decides to view the inventory, it shall contain all the DOC information and tomes of knowledge viewed up until that point. This is not necessary to progress through the game (and it cannot be done using a joystick controller), but it can be useful, as it helps players to recapitulate information from an earlier point that they may have forgotten.

3.5 Game Diagrams

In order to demonstrate the game's dynamics more easily (as well as provide a partial walk-through to the game itself), a diagram for each level was built.

The first level's diagram is represented by Figure 3.10, the second level's diagram is illustrated in Figure 3.11, the third level's diagram is presented by Figure 3.13a, the fourth level's diagram is illustrated in Figure 3.14, and the fifth level's diagram is represented by Figure 3.15.

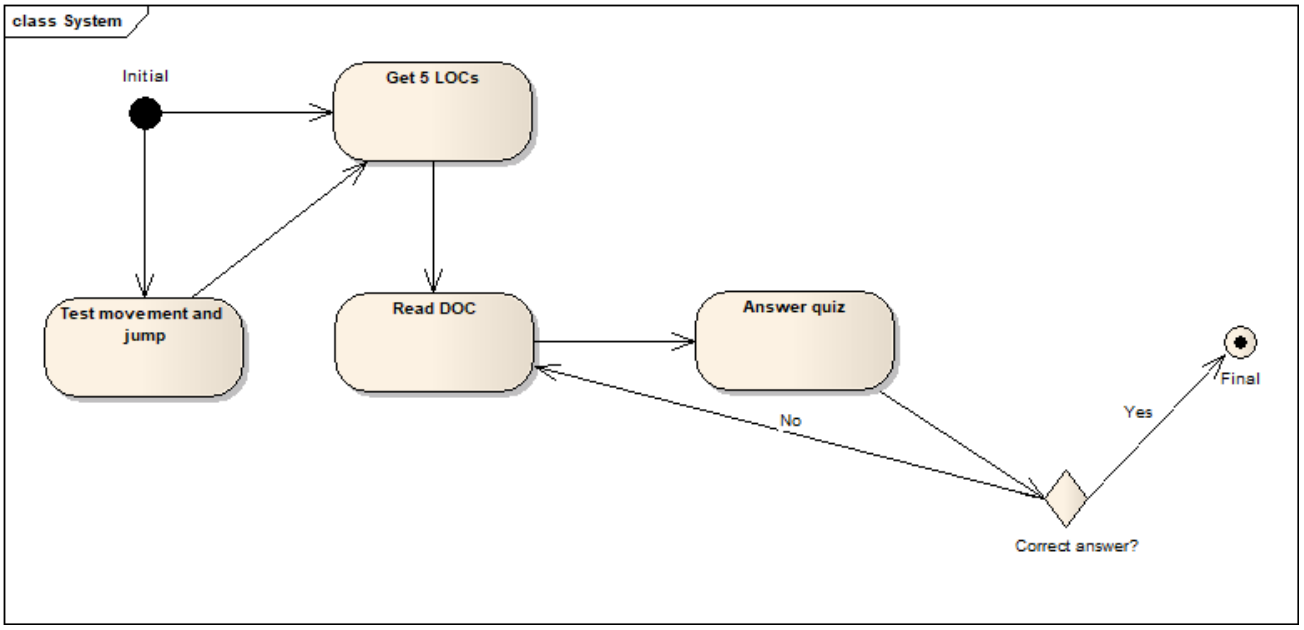


Figure 3.10: Level 1 Diagram

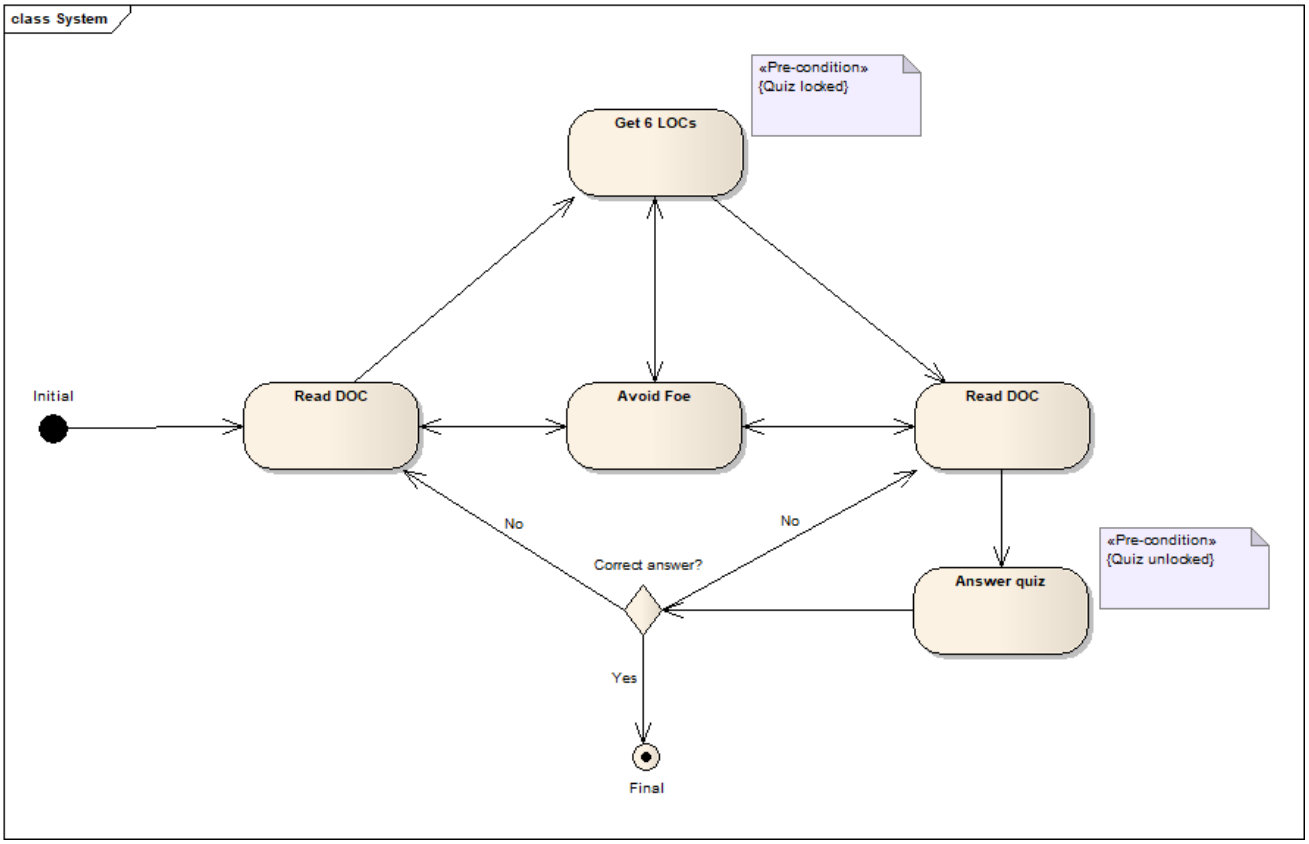


Figure 3.11: Level 2 Diagram

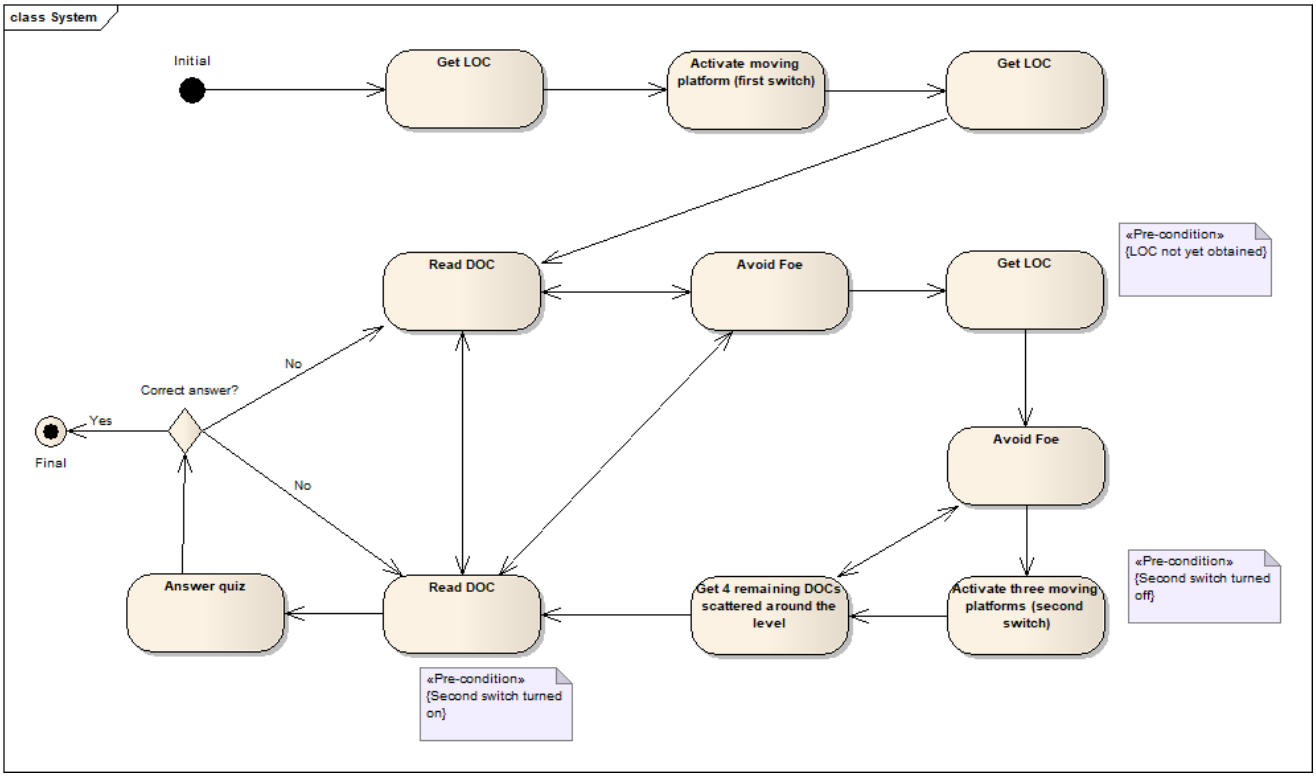


Figure 3.12: Level 3 Diagram

(a) Gaffe: 'Get 4 remaining LOCs', not 'DOCs'

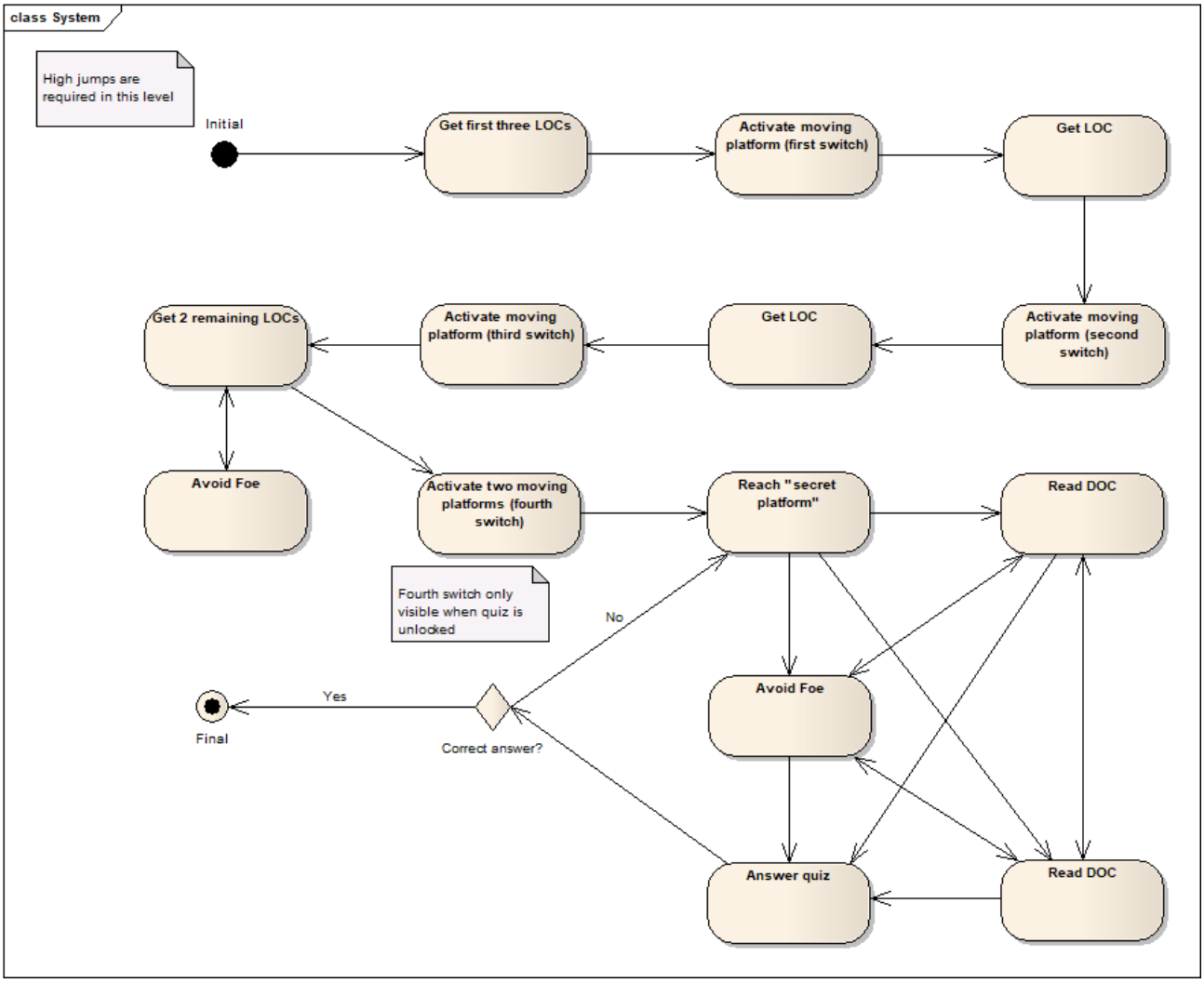


Figure 3.14: Level 4 Diagram

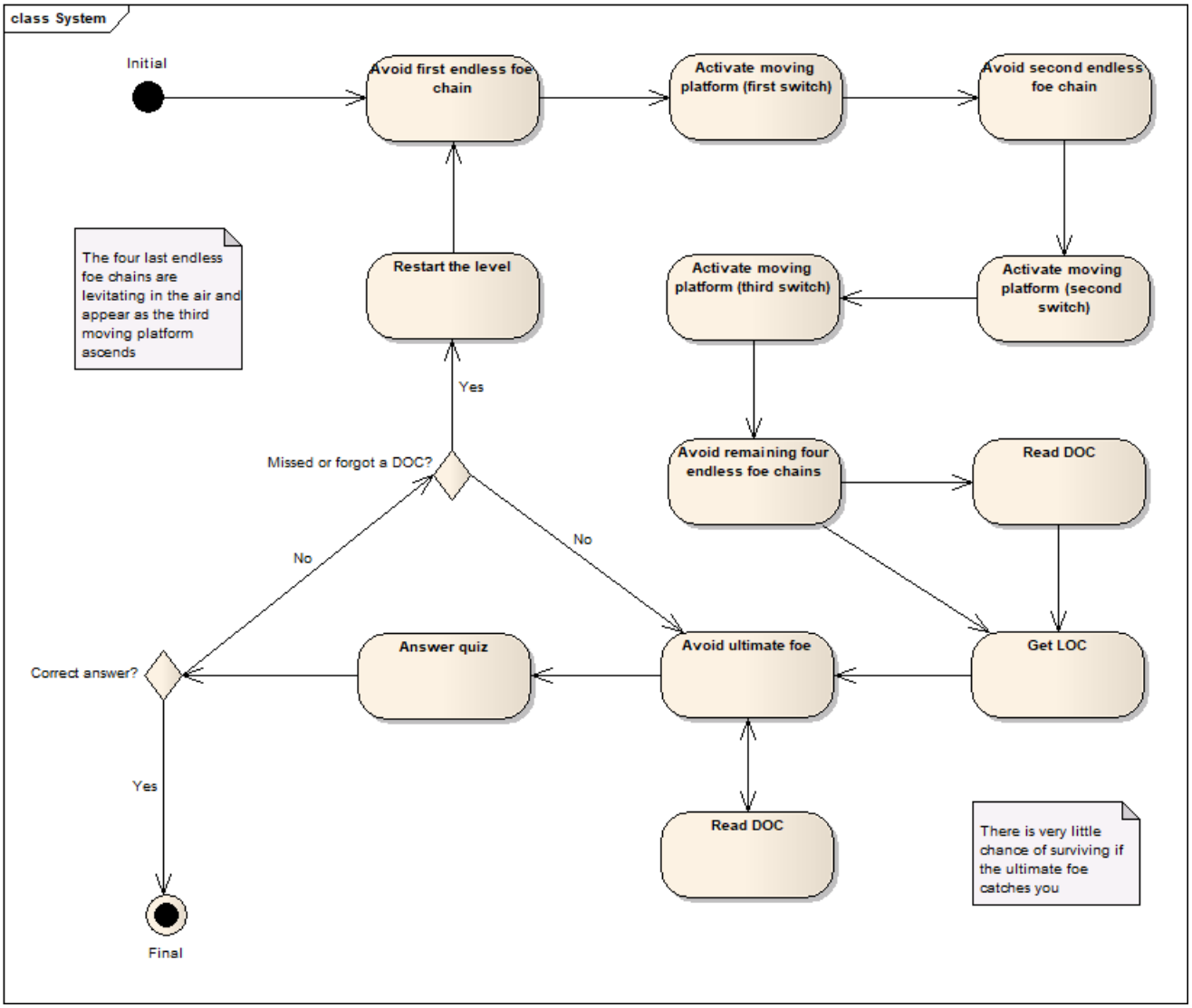


Figure 3.15: Level 5 Diagram

3.6 Game Data

As previously mentioned, the game is about the field of Software Architecture and Design and, therefore, it contains a summarized version of most of the software architecture concepts already detailed in the State of the Art:

Tome of Knowledge 1 'Software is a key component of modern technology. It is built into hardware such as computers, smartphones and other machines and, as such, it is responsible for the appropriate performance of said hardware. Naturally, it is a worthy affair to get to know about how software is organized in the current age, and the field that specifies in that matter is the field of software architecture.

Software Architecture and Design can be understood as a subtopic of the Software Engineering (SE) domain of knowledge. In itself, it is a domain area with a wide range of concepts

and knowledge with various possible applications, as well as a cornerstone for the life cycle of software development and, as such, essential for developers.

You're about to play a game designed to teach people about software architecture. Pay close attention to what you are about to learn, for it will be useful in the very near future.

Let's begin, shall we?'

Level 1 DOC 'Software architecture is a subtopic of the Software Engineering domain of knowledge and can be defined as the specification of the structure of a software system.'

Level 1 Quiz 'What is software architecture?

A: A subtopic of the Software Comprehension domain of knowledge that exists for mysterious reasons.

S: A subtopic of the Software Engineering domain of knowledge that can be defined as the specification of the structure of a software system.

D: A subtopic of the Software Engineering domain of knowledge that can be defined as the methodology of correction of semantic software errors. *(the correct answer is S)*'

Tome of Knowledge 2 'Why does a software system need to have its structure specified?

In order to understand that, we must need to look at the issues relevant to the design structure of software systems. As software systems become bigger and more complex over time, it becomes a necessity to solve the design problems that come with implementing these systems.

This is where software architecture comes in.'

Level 2 DOC 1 'Structural issues that software architecture is meant to address include the following:

Gross organization and global control structure;

Protocols for communication;

Synchronization and data access;

Assignment of functionality to design elements.'

Level 2 DOC 2 'Structural issues that software architecture is meant to address include the following:

Physical distribution;

Composition of design elements;

Scaling and performance;

Selection among design alternatives.'

Level 2 Quiz 'Which of the following answers represents a list of issues that software architecture is meant to address in its entirety?

U: Gross organization and global control structure; composition of design elements; effective debugging.

F: Protocols for communication; data cleansing; scaling and performance.

O: Synchronization and data access; physical distribution; selection among design alternatives. *(the correct answer is O)*'

Tome of Knowledge 3 'By planning and designing a robust structure for software systems using software architecture, we make sure that details within a software system such as the physical distribution and the protocols for communication are optimal, leading to more well-built software systems overall.

However, it's not enough to recognize that a software system needs to have a well-groomed structure. Rather, it's also important to be aware of the impact that said well-groomed structure carries on the system in question, not only in terms of its functionality but also in terms of how approachable it is from the perspective of the people who use it.'

Level 3 DOC 1 'Software architecture is important because it helps us understand large systems easily and it contributes to the reuse of large components and the frameworks in which these components can be integrated.'

Level 3 DOC 2 'The importance of software architecture is that it provides a partial blueprint for a software system's development by explicitly indicating its components and dependencies between them and it exposes the ways in which said system is expected to evolve.'

Level 3 Quiz 'Which one of these answer is TRUE?

I: Software architecture makes sure that none of the software system's components can be reused.

R: Software architecture proves that a software system is static, therefore proving its stability.

L: Software architecture makes it easier for a software system to be properly understood. *(the correct answer is L)*'

Tome of Knowledge 4 'Other reasons as to why software architecture matters are that it facilitates analysis of that system (through methods such as system consistency checking, conformance to quality attributes and dependence analysis, among others) and it contributes to the effective management of the system (through the grasping of requirements, implementation strategies and potential risks), leading to an impact in the market drivers relevant to software businesses.

We now know about what software architecture is and does, as well as its reason to exist. Now, we shall look at the various tools and practices that are involved in the development of a software architecture.

Beforehand, software architecture was largely an "ad-hoc" affair, where descriptions relied on box-and-line diagrams that were rarely maintained once a software system finished construction. Since then, more abstract factors relevant to the development of software architectures have been devised, and you are about to learn more about those factors right now...'

Level 4 DOC 1 'Architecture Description Languages (ADLs) are notations responsible for characterizing software architectures, typically also providing tools for parsing and generating architecture descriptions.

Distinct examples of ADLs include Adage, Aesop, C2, Darwin, Rapide, SADL, UniCon, Meta-H and Wright.'

Level 4 DOC 2 'There are different models (also known as styles or patterns) for architecture design. These models typically specify a design vocabulary, restrictions on how that vocabulary is employed and semantic assumptions about that vocabulary.

Distinct examples of architectural models include pipe-and-filter, blackboard, client-server, event-based and object-based.'

Level 4 Quiz 'Which one of these answers is FALSE?

I: Aesop, Darwin, SADL, UniCon, Meta-D and Wright are all valid examples of ADLs.

L: ADLs are responsible for characterizing software architectures and may also provide tools for parsing and generating software architecture descriptions.

U: Examples of software architecture models include the client-server model, the event-based model and the object-based model. *(the correct answer is I)*'

Tome of Knowledge 5 'Aside from ADLs and architectural models, other examples of factors integral to software architecture design are product lines and standards (where architectures are built to support a line of software systems as opposed to a solitary system), architecture viewpoints (which are sub-specifications meant for each of the multiple views each architecture is commonly organized into) and recovery methods each architecture carries. These factors are all put into practice while building software architectures.

Which leads us to one last question: what does the final result of that practice look like?

In the final leg of this journey, you shall learn about examples of software architectures, as well as characteristics that define each one of them.'

Level 5 DOC 1 'Some examples of software architectures are as follows:

The International Standard Organization's Open Systems Interconnection Reference Model, a layered network architecture;

The NIST/ECMA Reference Model, a generic software engineering environment architecture based on layered communication substrates;

The X Window System, a distributed windowed user interface architecture based on event triggering and callbacks.'

Level 5 DOC 2 'Another example of a software architecture is Camelot, which is built using the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and servers.'

Level 5 Quiz 'Which of these answers refers to a software architecture that uses remote procedure calls both locally and remotely?

D: Camelot

I: X Window System

E: NIST/ECMA Reference Model *(the correct answer is D)*'

Tome of Knowledge 6 'There are other examples of software architectures, such as the High Level Architecture for Distributed Simulation (an architecture that allows the integration of simulations provided by vendors and operates based on international cross-vendor integration standards) and the Sun's Enterprise Java Beans Architecture (an architecture that supports distributed, Java-based, enterprise-level applications and operates based on ad-hoc cross-vendor standards defined by Java's industrial leaders).

You have reached the end of the game. We hope you were taught some useful information. However, know that if you are interested in mastering the subject of software architecture, this game is only the beginning.'

3.7 Screenshots

These are screenshots from certain points of the game, taken to help the readers of this thesis have a better idea of what the finished product looks like.

The first screenshot is of the main menu (Figure 3.16).

Figure 3.16: Main Menu

The second screenshot shows a still from the game's first level (Figure 3.17).

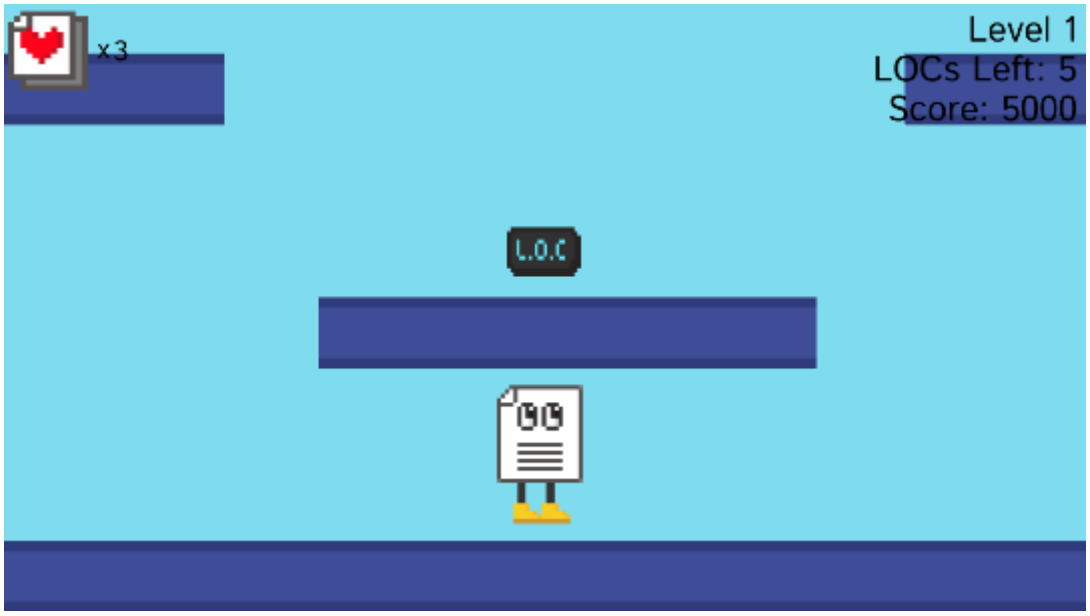


Figure 3.17: First Level

The third screenshot shows one of the game's quizzes, more specifically the one from Level 2 (Figure 3.18).

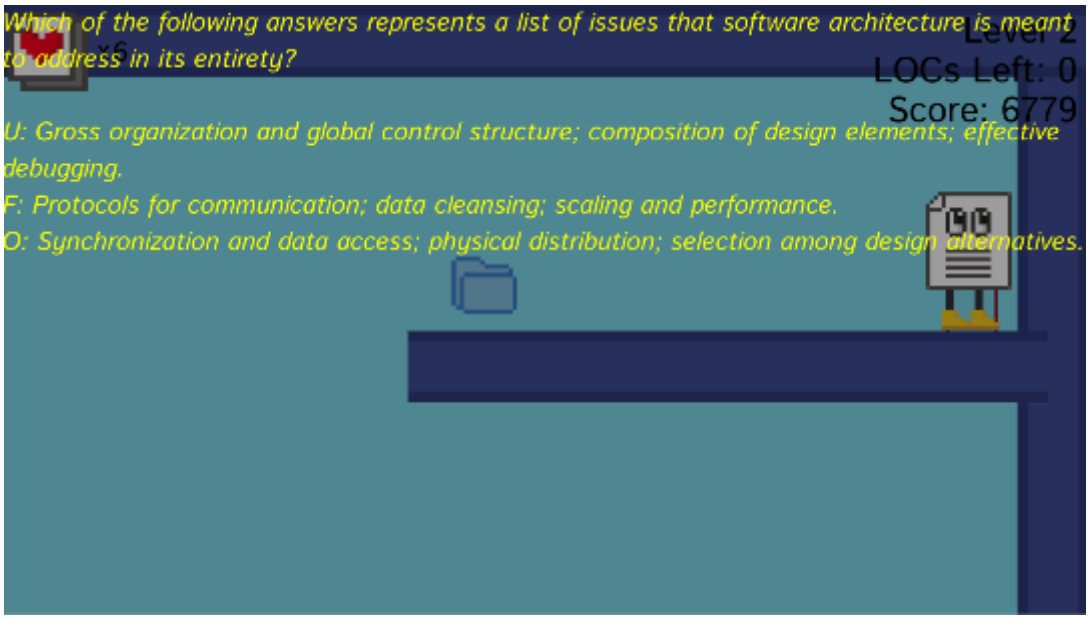


Figure 3.18: Quiz

The fourth and final screenshot shows an example of a tome of knowledge (Figure 3.19).

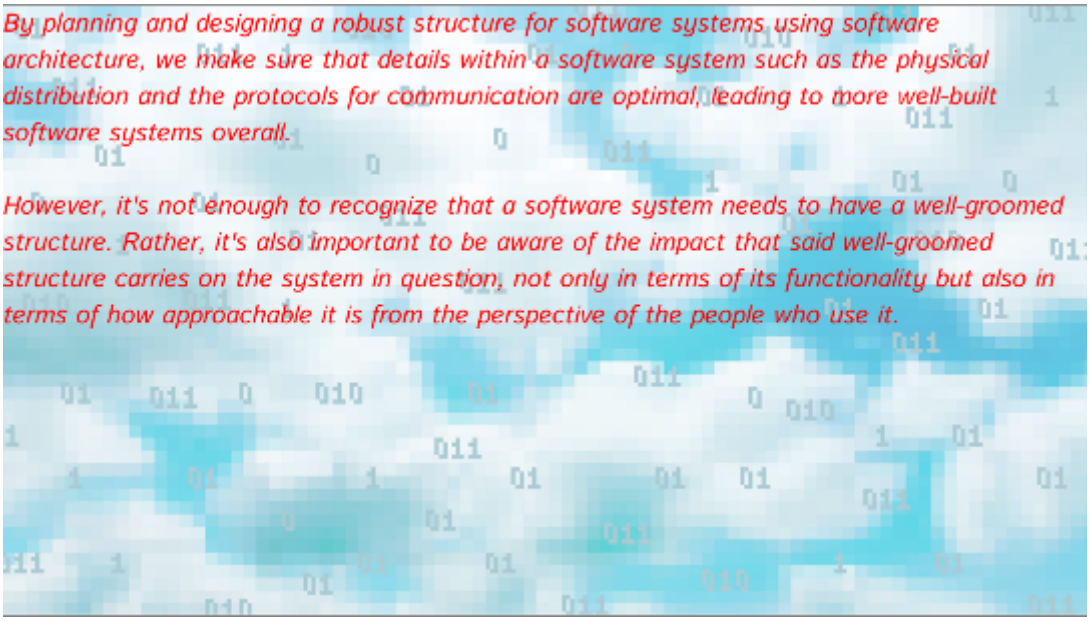


Figure 3.19: Tome of Knowledge

3.8 Development Methodology

The software development model largely used during the game's development is Barry Boehm's 'Spiral Model' (illustrated in Figure 3.20), which is divided in four phases:

1. Determination of objectives;

2. Identification and resolution of risks;

3. Development and test;

4. Planning of the next iteration (if project is not yet complete).

In other words, it is a model that emphasizes a 'test-first' strategy - where the complete planned work is divided into fragments and each fragment of the work is done and then tested individually, and work on the next fragment only begins when the testing of the first fragment produces successful results.

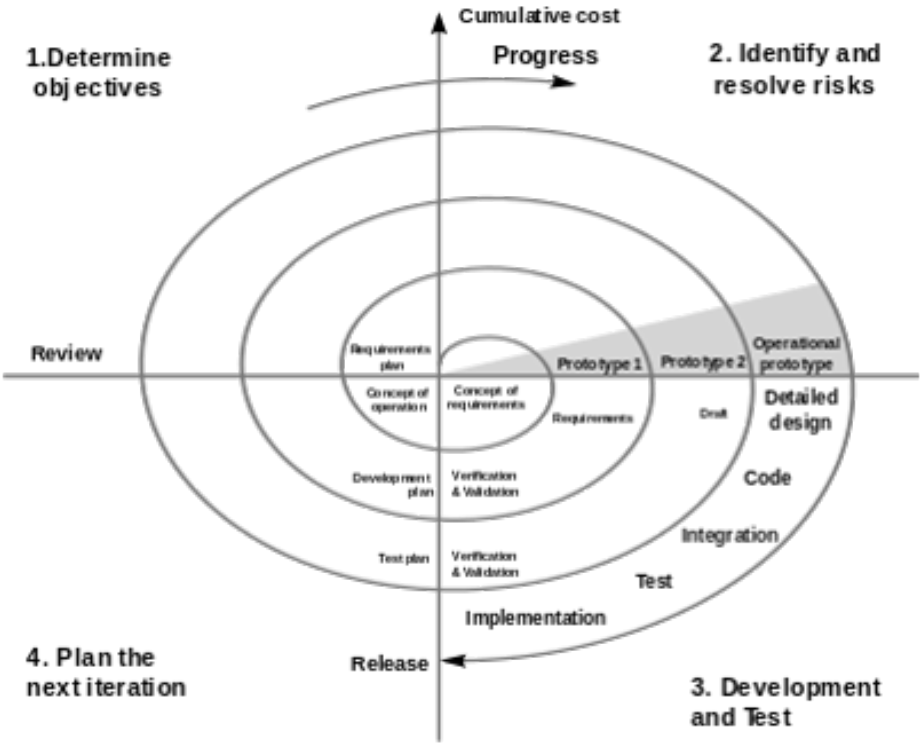


Figure 3.20: The Spiral Model (Barry Boehm, 1988)

3.9 Summary

There are issues in the traditional software engineering education models.

An attempt to solve these issues was made by developing a serious game to teach people about software architecture and design.

The serious game is a 2D platformer called 'Codebase Escape', made using Unity and C#.

The players must clear 5 levels by collecting all the LOCs, viewing all the DOCs and answering the quiz correctly in each level.

The faster the players can clear the game, the higher score they will get.

The game has foes, and if the main character touches them, it will carry a negative impact on its movement speed, jump power and/or HP count.

If the main character loses all its HPs, the game is over.

Each LOC weighs down the main character, and to compensate for this, clearing each quiz will give the main character certain stat boosts.

The game was developed using Barry Boehm's 'Spiral Model'.

The game was developed from late February to the end of May, the ESWS that validated it was done on the 1st of June, and the dissertation was written and verified during the rest of June.

Chapter 4

Empirical Study With Students

The game was validated through an empirical study with students (***ESWS***). Here is how it went down chronologically:

1. A proposal to participate in an experiment for a dissertation was sent to MIEIC's 1st-year students, and a total of 15 students decided to participate;
2. Then, on the day the ESWS started, the students that participated were screened for knowledge on the topic of SAD through a background quiz (***BG***);
3. The students were then divided into two groups: Group A (with 8 students) and Group B (with 7 students);
4. Each of the groups was put in a different room, where Group A played the game and Group B answered the knowledge quiz (***KW***) without having played the game beforehand;
5. Then, after having played the game, Group A proceeded to answer the same KW that Group B did;
6. In addition to that, Group A also answered an external factors quiz (***EF***) and an overall satisfaction quiz (***OS***), for the purposes of ruling out threats to validity related to the game's quality and the overall atmosphere that surrounded the study;
7. After the experiment, the results to all the quizzes were collected and analyzed. Three test subjects had their KW results discarded because of their high BG results, considering the purpose was to get students with initially reduced knowledge in SAD (to clarify, the students had to score below 3 in the BG to have their KW results count, a requirement those three test subjects failed to meet).
8. Finally, the KW results of all the other subjects were collected and, out of those, the KW results that belonged to Group A members were compared to the KW results that belonged to Group B members.

The preferred outcome of the experiment is that Group A performs better at the questionnaire than Group B, since it was Group A that played the game. Was that outcome achieved? In order to determine that, a closer look at this experiment is necessary.

For the BG, EF and OS quizzes, a classification in the style of the Likert scale is used, meaning that the questions are asked in the form of affirmations that the user can either agree with, disagree with or stand in the middle on. A 1-5 scale is used in this type of classification, where '1' means 'strongly disagree', '2' means 'somewhat disagree', '3' means 'neither agree nor disagree', '4' means 'somewhat agree' and '5' means 'strongly agree'.

The results for the BG, the EF and the OS are all calculated in a similar fashion:

- All the answers (1-5) are collected and inserted in tables;
- In the process of calculating the averages, each answer to a question that reflects lack of knowledge or experience or satisfaction is inverted (even if the answers themselves stay unaltered);
- For each student, the average for each of quizzes is calculated to reflect their individual performance;
- Finally, the average of all the answers for a specific question is also calculated to reflect how the test subjects as a whole handled that question.

4.1 The Background Quiz

- BG1-4. I have considerable knowledge on...
  - ...programming languages.
  - ...software architecture.
  - ...software design.
  - ...developing software.
- BG5. I've never played any game that enabled me to learn about Software Architecture.

4.1.1 Results

BG's results are shown in Table 4.1:

BG Results	BG1	BG2	BG3	BG4	BG5	Average
A1	3	2	1	2	5	1.8
A2	2	1	1	1	5	1.2
A3	3	1	1	2	5	1.6
A4	3	3	2	1	5	2
A5	3	3	3	3	1	3.4
A6	3	3	1	1	2	2.4
A7	4	4	4	4	1	4.2
A8	3	1	1	2	5	1.6
B1	3	2	1	2	5	2
B2	3	2	1	2	5	3
B3	3	2	1	2	5	2.6
B4	3	2	1	2	5	2.4
B6	3	2	1	2	5	1.6
B7	3	2	1	2	5	2.2
B8	3	2	1	2	5	2.6
Avg. A	3	2.3	1.8	2	2.4	
St. Dev. A	0.5	1.1	1.1	1	1.8	
Avg. B	3.1	2.4	2.1	2.6	1.4	
St. Dev. B	0.6	0.9	0.8	0.9	1	

Table 4.1: Background Quiz Results

According to the results, we can see that the three previously mentioned test subjects whose KW results were ignored were A5, A7 and B2. On the whole, however, the background quiz results are satisfactory: the test subjects had a mild knowledge on programming languages, but limited knowledge on software architecture, design and development. Most of the test subjects also never played a game that enabled them to learn about software architecture.

4.2 The External Factors Quiz

- EF1. I found the whole experience intidimating.
- EF2. I like to play computer games.
- EF3. I was quite tired when I began to play the game.
- EF4. I kept getting distracted by other colleagues.

4.2.1 Results

EF's results are shown in Table 4.2:

EF Results	EF1	EF2	EF3	EF4	Average
A1	1	5	4	1	4.3
A2	1	5	4	1	4.3
A3	1	5	3	1	4.5
A4	5	5	2	1	3.8
A5	2	4	4	3	3.3
A6	1	5	2	2	4.5
A7	4	5	2	1	4
A8	1	4	1	1	4.8
Avg.	4	4.8	3.3	4.6	
St. Dev.	1.5	0.4	1.1	0.7	

Table 4.2: External Factors Quiz Results

The experiment was largely successful in providing an inviting atmosphere and environment to the test subjects. The only significant *caveat* was that the experiment took place right after a mini-test the test subjects did, which was reflected in the results to EF3.

4.3 The Overall Satisfaction Quiz

- OS1. I had fun playing the game.
- OS2. I found the game to be balanced in terms of being 'hard to play'.
- OS3. I liked the visual aspect of the game.
- OS4. I liked the audio environment inside the game (only for those who heard the audio).
- OS5. I feel I have learned new and valuable information.
- OS6. I felt the amount of information inside the game to be overwhelming.
- OS7. I would play this game again.

4.3.1 Results

OS' results are shown in Table 4.3:

OS Results	OS1	OS2	OS3	OS4	OS5	OS6	OS7	Average
A1	3	4	3	4	3	5	3	3
A2	3	3	4	3	2	5	2	2.6
A3	4	3	4	4	3	5	3	3.1
A4	1	2	1		3	1	2	2.3
A5	2	2	2		2	3	2	2.2
A6	4	3	3	3	4	3	3	3.3
A7	5	3	4	4	2	4	4	3.4
A8	1	3	2	2	2	4	1	1.9
Avg.	2.9	2.9	2.9	3.3	2.6	2.3	2.5	
St. Dev.	1.4	0.6	1.1	0.7	0.7	1.3	0.9	

Table 4.3: Overall Satisfaction Quiz Results

When it comes to the game itself and the overall satisfaction the test subjects had with it, the results were somewhat lukewarm. The most prominent complaint was directed at the amount of information within the game being somewhat overbearing (most likely referring to the tomes of knowledge and their 'wall of text' format), which affected the learning process and the replayability. The visuals had a mixed reception and the audio was received mildly positively.

4.4 The Knowledge Quiz

- **KW1. What is Software Architecture?**

  Solution: The specification of the structure of a software system (and also a subtopic of the Software Engineering domain of knowledge).

- **KW2. Give four examples of issues that Software Architecture is meant to address.**

  Solution: Citing or referring to at least four out of the following: gross organization and global control structure, protocols for communication, synchronization and data access, assignment of functionality to design elements, physical distribution, composition of design elements, scaling and performance, selection among design alternatives.

- **KW3. Of the above issues, why did it become necessary, over time, to solve them?**

  Solution: Because, as time passed, software systems became bigger and more complex, leading the possible amount of issues to become larger.

- **KW4. Software Architecture... (1-5, from false to true)**

  - **KW4.1. ...contributes to more well-built software overall.** Solution: True.

  - **KW4.2. ...openly indicates the components of software systems and dependencies between them.** Solution: True (It is by doing this that software architecture provides a partial blueprint for the development of software systems).
  - **KW4.3. ...makes software systems more secure by preventing their components from being used more than once.** Solution: False (Software architecture contributes to the reuse of large components and the frameworks in which these components can be integrated).
  - **KW4.4. ...helps users understand software systems, to the point where analysis of these systems becomes unnecessary.** Solution: False (It is true that software architecture helps users understand software systems, however, this does not make the analysis of said systems unnecessary. Rather, software architecture also facilitates analysis of software systems through various methods).
  - **KW4.5. ...helps users grasp the requirements and potential risks of software systems.** Solution: True (It is by doing this that software architecture manages to contribute to the effective management of software systems).
  - **KW4.6. ...makes software systems more stable by making them static.** Solution: False (Software architecture does make software systems more stable, but it does not impede the evolution of software systems; rather, it exposes the way these systems are expected to evolve).

- **KW5. Box-and-line diagrams and architecture viewpoints are both factors to the current development of software architectures, as opposed to the "ad-hoc" affair that once described the software architecture development. Is this true or false? Explain why.**

  Solution: False. Architecture viewpoints are a factor to current software architecture development, but box-and-line diagrams are a factor to older, "ad-hoc" software architecture development.

- **KW6. Give, at least, three examples of Architecture Description Languages (ADLs).**

  Solution: Citing at least three out of the following: Adage, Aesop, C2, Darwin, Rapide, SADL, UniCon, Meta-H, Wright.

- **KW7. Give, at least, three examples of Architecture Models/Styles/Patterns**

  Solution: Citing at least three out of the following: pipe-and-filter, blackboard, client-server, event-based, object-based.

- **KW8. *Camelot*, *X Window System*, *NIST/ECMA Reference Model*. Which of these refers to a software architecture that is based on event triggering and callbacks?**

  X Window System (Not to be confused with Level 5's quiz, which asks which of these refers to a software architecture that uses remote procedure calls both locally and remotely, the correct answer there being Camelot).

- **KW9. What do you consider to be the main difference between the High Level Architecture for Distributed Simulation and the Sun's Enterprise Java Beans Architecture?**

  Solution: The former has international application support and cross-vendor standards, while the latter has enterprise-level application support and cross-vendor standards (more specifically standards defined by Java's industrial leaders).

- **KW10. (Yes or No) Do you think the SOLID mnemonic has anything to do with software architecture?**

  Solution: Yes (The SOLID mnemonic refers to the set of principles used in object-oriented software design. S – Single Responsibility, O – Open-closed, L – Liskov Substitution, I – Interface Segregation, D – Dependency Inversion. The game's structure is based on this mnemonic, with each level representing a principle that gets applied to the anthropomorphic code when the player clears a quiz. It is also related to the game's 'easter egg', where each of the five letters refers to the key the player has to press to pick the correct answer to the quiz of each level, respectively (which is also the hint to the correct answer to this question)).

4.4.1 Results

The results of the KW, which also use the 1-5 classification tactic, were calculated as such:

- The criteria used to classify answers to the average open-answer KW question is based on how close each test subject is to the correct answer:
  - 1 - The subject answered incorrectly;
  - 2 - The subject answered somewhat incorrectly;
  - 3 - The subject did not answer (or it is ambiguous whether answer is correct or incorrect);
  - 4 - The subject answered somewhat correctly;
  - 5 - The subject answered correctly.
- The criteria used to classify answers to each of the questions in KW4 was as follows:
  - The exact correct answer to true statements was 5, while the exact correct answer to false statements was 1;
  - Therefore, the closer the subject gets to answering a true statement with a 5, or to answering a false statement with a 1, the better the subject's score on these questions will be.
- The criteria used to classify answers to KW5 is as follows:
  - 1 - Incorrect answer without a justification or a valid attempt at one;
  - 2 - Incorrect answer with a justification;

  - 3 - Either no answer or correct answer without a justification;
  - 4 - Correct answer with a poor justification;
  - 5 - Correct answer with a solid justification.

- The criteria used to classify answers to the KW's closed-answer questions (KW8 and KW10) is as follows:
  - 1 - The subject answered incorrectly;
  - 3 - The subject did not answer;
  - 5 - The subject answered correctly.

KW's results are shown in Table 4.4a.

KW Results	**KW 1**	**KW 2**	**KW 3**	**KW 4.1**	**KW 4.2**	**KW 4.3 (a)**	**KW 4.4 (a)**	**KW 4.5**	**KW 4.6 (a)**	**KW 5**	**KW 6**	**KW 7**	**KW 8**	**KW 9**	**KW 10**	**Avg.**
A1	5	5	4	5	4	1	3	3	1	5	3	3	5	4	5	4.3
A2	3	3	3	5	3	3	3	4	2	3	3	3	3	3	1	3.1
A3	3	3	1	4	3	3	2	4	2	1	1	3	1	3	5	2.9
A4	4	3	5	4	3	2	1	1	4	3	3	3	3	3	5	3.4
A6	4	1	3	5	4	2	3	3	1	1	3	2	1	3	5	3.1
A8	4	3	4	5	5	1	4	1	3	2	1	3	1	3	3	3
B1	5	3	3	4	3	1	4	2	1	3	3	1	3	3	5	3.3
B3	3	3	3	5	3	3	3	3	3	3	3	3	3	3	5	3.3
B4	2	5	1	4	4	4	3	3	2	3	3	1	1	3	5	2.9
B6	4	3	3	5	4	3	4	3	3	3	3	3	3	3	3	3.2
B7	3	3	3	4	3	3	4	4	3	3	3	3	3	3	3	3.1
B8	3	4	2	5	4	4	2	2	4	3	3	3	3	3	5	3.2
Avg. A	**3.8**	**3**	**3.3**	**4.7**	**3.7**	**2**	**2.7**	**2.7**	**2.2**	**2.5**	**2.3**	**2.8**	**2.3**	**3.2**	**4**	
Avg. B	**3.3**	**3.5**	**2.5**	**4.5**	**3.5**	**3**	**3.3**	**2.8**	**2.7**	**3**	**3**	**2.3**	**2.7**	**3**	**4.3**	
St. Dev. A	0.7	1.2	1.2	0.5	0.7	0.8	0.9	1.2	1.1	1.4	0.9	0.4	1.5	0.4	1.5	
St. Dev. B	0.9	0.8	0.8	0.5	0.5	1	0.7	0.7	0.9	0	0	0.9	0.7	0	0.9	

Table 4.4: Knowledge Quiz Results

(a) Question where Group A's average is actually supposed to be lower

Looking at the results, most of the test subjects in both groups had an average KW score. However, the initial goal of this study was to make it so that Group A, as a whole, would have a better average at the KW quiz than Group B, considering that Group A had their results after playing the game and Group B had their results before playing the game. To an extent, the study could be considered successful, as Group A performed better than Group B on most questions (especially the earlier ones), meaning the game did its job somewhat well when it came to these test subjects.

When it comes to the questions where Group B outperformed, however, it is possible that some of the students who played the game might have misinterpreted the information present in the game. For instance, in KW8, a good amount of Group A's test subjects confused the question with a question from the game that seemed somewhat similar, leading them to write the correct answer to the game's question, as opposed to the correct answer to KW8. This led to the Group B subjects having an advantage, since they mostly did not answer and, therefore, managed to score higher. The game's somewhat mixed reception might have also played a part in some Group A subjects getting answers wrong: since they did not enjoy the game and the way it was structured that much, it is possible that, as the game became tougher and more complex as it went on, they did not feel motivated enough to remember the game's information all that well. This could also explain how the Group A subjects left some of the later questions unanswered.

There are also other factors to consider, such as the possibility that the test subjects who played the game chose to ignore the incentives to try to answer quizzes and, instead, decided to rely on their luck rather than reading the DOCs; the possibility that test subjects (especially on Group B) guessed the answers to questions they don't know about and accidentally answered these questions correctly; and exceptions to the norm, the most notable being A1's stellar KW score, especially when compared to the average scores from the other test subjects.

Additionally, we can conclude that the test subjects had relatively similar standards to one another, given that the highest standard deviation in all quiz result tables shown is 1.5, which is not decidedly a high value.

For the sake of completion, it is also worth mentioning that Group B later went on to play the game and proceeded to answer the KW again (as well as the EF and the OS), but these actions were not considered for the ESWS.

4.5 Summary

An experiment with students of two groups was done, where one group played the game and the other group did not.

Both groups answered a background quiz. The results of this quiz were satisfactory for the most part, with only three test subjects falling outside of expectations.

The first group answered the knowledge quiz after playing the game, whereas the second group answered it before playing the game. The first group also answered quizzes related to external factors and overall satisfaction after playing the game.

The EF's results were very satisfactory as a whole.

The results of the OS and the KW, on the other hand, were more lukewarm, for a variety of different possible reasons.

Because the first group ended up performing better at the KW on average than the second group, the experiment was considered a mild success.

Chapter 5

Conclusion

As software's performance and prominence continues to increase over time, getting to know more about how software operates becomes more and more essential to people. Software Architecture and Design, an area within Software Engineering, contributes to this knowledge in a significant way. Part of what makes software systems function as well as they do is their structure and organization and, after doing extensive research around this topic, it can definitely be stated that there is a sizable number of different scenarios to consider and tasks to fulfill while building an architecture for a software system.

In addition to that, it is of noted importance that new and engaging methods of teaching subjects such as SAD are encouraged and supported. Even if ***Codebase Escape*** may not have been the most engaging educative game out there, it still contributed to opening the door to these methods by the way of combining LTFs with GDPs and containing somewhat intricate level design, as well as a lot of decently-implemented basic information about SAD that people can learn from.

With all that established, the research around SAD still continues. The main suggestions for future work all involve enhancing the game itself, with critiques raised by the empirical study's test subjects properly addressed. This includes building a better user interface around the tomes of knowledge in order to make them more than just 'walls of text' (so that they do not break up the game's pace); presenting information in new ways that would be easier for the players to remember; building stronger incentives for the players to answer the quizzes the intended way (in other words, by reading the DOCs beforehand, as opposed to trying to answer them by luck); and perhaps improving the game's aesthetics and difficulty balance.

Either way, we hope that this project and dissertation have made a noticeable contribution to the fields of Software Architecture and Design and Serious Games.

Bibliographical References

[ABK10] P. Abrahamsson, M. A. Babar, and P. Kruchten. Agility and Architecture: Can They Coexist? *IEEE Software*, 27(2):16–22, February 2010. Software architecture.

[Ben] F. Benitti. SE RPG. http://www.inf.ufsc.br/~fabiane.benitti/serpg/. Last accessed in 2017-06-17.

[BH04] S. Bjork and J. Holopainen. *Patterns in Game Design*. Charles River Media Game Development. Charles River Media, 1st edition, December 2004. Software design.

[Coe15] A. Coelho. Gamification. FEUP Presentation, November 2015. Game design.

[Cru10] N. Cruz. Jogo Sério para Aprendizagem de Estimação em Projetos de Software. Master's thesis, Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, 4200-465 Porto, July 2010. Software engineering.

[Far16] R. Faria. Game Design Techniques for Software Engineering. Master's thesis, Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, 4200-465 Porto, July 2016. Software design.

[FMCS12] V. Farias, C. Moreira, E. Coutinho, and I. Santos. iTest Learning: Um Jogo para o Ensino do Planejamento de Testes de Software. In *V Fórum de Educação em Engenharia de Software*, Natal, Brazil, September 2012. FEES. Software testing.

[FS10] J. Fernandes and S. Sousa. PlayScrum - A Card Game to Learn the Scrum Agile Method. Master's thesis, Universidade do Minho, R. da Universidade, 4710-057 Braga, March 2010. Software engineering.

[Gar00] D. Garlan. Software architecture: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 91–101, Limerick, Ireland, June 2000. ICSE, ACM New York. Software architecture.

[Gro07] M. Grosser. Effective teaching: linking teaching to learning functions. *South African Journal of Education*, 27(1):37–52, February 2007. Software design.

[GS93] D. Garlan and M. Shaw. An Introduction to Software Architecture. In S. K. Chang, V. Ambriola, and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 2 of *Series on Software Engineering and Knowledge Engineering*, chapter 1, pages 1–40. World Scientific Publishing Co. Pte. Ltd., P.O. Box 128, Farrer Road, Singapore 9128, December 1993. Software architecture.

[kn:a] Educational game for software measurement training X-MED (in portuguese). http://www.gqs.ufsc.br/educational-game-for-software-measurement-training-x-med-in-portuguese/ Last accessed in 2017-06-17.

[kn:b] SESAM: Promoting Simulation in Medical Education. https://www.sesam-web.org/. Last accessed in 2017-06-17.

[LPF15] P. Letra, A. Paiva, and N. Flores. Game Design Techniques for Software Engineering Management Education. Master's thesis, Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, 4200-465 Porto, October 2015. Software design.

[Mar02] R. Martin. *Agile Software Development: Principles, Patterns and Practices*. Pearson Education, Inc., Upper Saddle River, New Jersey 07458, 1st edition, October 2002. Software development.

[May05] N. May. A Survey of Software Architecture Viewpoint Models. In J. Schneider, editor, *The Sixth Australasian Workshop on Software and System Architectures*, pages 13–24, Brisbane, Australia, March 2005. AWSA. Software architecture.

[MT00] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000. Software architecture.

[MT10] N. Medvidovic and R. N. Taylor. Software architecture: foundations, theory and practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 2, pages 471–472, Cape Town, South Africa, May 2010. ICSE, ACM New York.

[N+10] E. Navarro et al. SimSE: An Educational, Game-Based Software Engineering Simulation Environment. http://www.ics.uci.edu/~emilyo/SimSE/index.html, 2010. Last accessed in 2017-06-17.

[Pan10] R. K. Pandey. Architecture Description Languages (ADLs) vs UML: A Review. *ACM SIGSOFT Software Engineering Notes*, 35(3):1–5, May 2010. Software architecture.

[PW92] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992. Software architecture.

BIBLIOGRAPHICAL REFERENCES

[RA07] G. Rasool and N. Asif. Software Architecture Recovery. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 1(4):939–944, 2007. Software architecture.

[Rib14] T. Ribeiro. iLearnTest: Jogo Educativo para Aprendizagem de Testes de Software. Master's thesis, Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, 4200-465 Porto, July 2014. Software testing.

[SB11] L. Silva and D. Balasubramaniam. Controlling Software Architecture Erosion: A Survey. *Journal of Systems and Software*, 85(1):132–151, July 2011. Software architecture.

[SKA15] A. Sharma, M. Kumar, and S. Agarwar. A Complete Survey on Software Architectural Styles and Patterns. In A.K. Singh and J. Mathew, editors, *Procedia Computer Science*, volume 70 of *Proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems*, pages 16–28, Kurukshetra, India, November 2015. ICECCS, Elsevier B. V. Software architecture.

[TZG10] M. Thiry, A. Zoucas, and A. Gonçalves. Promovendo a Aprendizagem de Engenharia de Requisitos de Software Através de um Jogo Educativo. In *Simpósio Brasileiro de Informática na Educação*. SBIE, 2010. Software design.

[U$^+$] C. Ungashick et al. uTest - The Professional Network for Testers. https://www.utest.com/. Software testing, last accessed in 2017-06-17.

[Var11] M. Varshney. SIM.JS - Discrete Event Simulation in JavaScript. http://simjs.com/, 2011. Last accessed in 2017-06-17.

Appendix A

Project Chronology

The chronology of this project was as follows:

- In late **February**, when the second semester started, the development of the game officially began;
- The development of the game went on from **March** to **May**, where meetings between the student-author and the supervisor were arranged on a two-week basis to suggest new ideas and plan further work;
- In late **May**, the empirical study with students, which will be talked about in a moment, was devised;
- The empirical study with students was finally done on the 1st of **June**;
- Finally, during the rest of **June**, the dissertation was written. The technical report that was finished for Dissertation Planning was used as a template for the dissertation itself.